Example-based Rendering of Textural Phenomena

A Thesis
Presented to
The Academic Faculty

by

Vivek Kwatra

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

College of Computing
Georgia Institute of Technology
August 2005

Example-based Rendering of Textural Phenomena

Approved by:

Dr. Aaron Bobick, Advisor
College of Computing
Georgia Institute of Technology

Dr. Jarek Rossignac
College of Computing
Georgia Institute of Technology

Dr. Irfan Essa, Co-Advisor
College of Computing
Georgia Institute of Technology

Dr. Steven Seitz
Department of Computer Science and Engineering
University of Washington at Seattle

Dr. Ramesh Raskar
Mitsubishi Electric Research Labs

Dr. Greg Turk
College of Computing
Georgia Institute of Technology

Date Approved: 21 June 2005

In loving memory of my father, Dr. Suresh Kumar Kwatra, who has been my greatest inspiration in life

ACKNOWLEDGEMENTS

I am indebted to my advisor, Aaron Bobick, and co-advisor, Irfan Essa, for their exceptional support and guidance. They gave me the freedom to explore research ideas at will, but were always there to educate and guide me. I would also like to thank the rest of my committee members including Greg Turk, Jarek Rossignac, Ramesh Raskar, and Steve Seitz for their excellent insights and advice.

I am grateful to my friends and colleagues for making the last six years at Georgia Tech immensely enjoyable. I would especially like to thank members of the Computational Perception Lab, the Geometry Group, and the Wall Lab. I sincerely cherish my engaging discussions with the ever helpful Gabriel Brostow, and collaboration with the insightful Arno Schödl. I thank Drew Steedly, Antonio Haro, Amos Johnson, Rawesak "Tee" Tanawongsuwan, Eugene Zhang, Stephanie Brubaker, Raffay Hamid, Mike Terry, Delphine Nain, Brooks Van Horn, Mark Carlson, and Jerry Choi for the many fruitful interactions I have had with them. I also thank Ravi Ruddarraju, Yifan Shi, Gaurav Chanda, Mitch Parry, Jie Sun, Pei Yen, Charlie Brubaker, Samir Batta, Nick Diakopoulos, Yan Huang, Siddhartha Maddi, and Howard Zhou for their help and support.

I would like to express my gratitude towards the GVU office staff including but not limited to Joan Morton, Chrissy Hendricks, Wanda Abbott, Jacquelyn Berry, Joi Adams, Leisha Chappell, Vivian Chandler, and David White, as well as Barbara Binder and Becky Wilson at the CoC, who have taken utmost care of me all these years. I also thank Spencer Reynolds, Jonathan Shaw, and Steve Park, who have been outstanding GVU Lab managers, as well as Peter Wan, Bernard Bomba-Ire, Pam Buffington, Karen Carter, and the rest of CNS for providing excellent and prompt systems support.

I have a deep regard for my wife, Aditi, who has always been supportive of me and

provided me with immeasurable love and inspiration. I am profoundly thankful to my parents for creating all the opportunities for me and for their love, affection, and encouragement – my mother's determination and father's vision have been of great significance in my life. I thank my grandmother for her blessings that have been with me at every step. My brother, Nipun, has been a great friend as well as colleague, and it was a great experience to collaborate with him on research, which I hope will continue. I would like to thank my brother-in-law, Aditya, for his extensive support and friendship. I also thank my parents-in-law for providing me with abundant love, care, and valuable guidance all these years.

TABLE OF CONTENTS

LIST OF FIGURES

SUMMARY

This thesis explores synthesis by example as a paradigm for rendering real-world phenomena. In particular, phenomena that can be visually described as texture are considered. We exploit, for synthesis, the self-repeating nature of the visual elements constituting these texture exemplars. Techniques for unconstrained as well as constrained/controllable synthesis of both image and video textures are presented.

For unconstrained synthesis, we present two robust techniques that can perform spatio-temporal extension, editing, and merging of image as well as video textures. In one of these techniques, large patches of input texture are automatically aligned and seamless stitched with each other to generate realistic looking images and videos. The second technique is based on iterative optimization of a global energy function that measures the quality of the synthesized texture with respect to the given input exemplar.

We also present a technique for controllable texture synthesis. In particular, it allows for generation of motion-controlled texture animations that follow a specified flow field. Animations synthesized in this fashion maintain the structural properties like local shape, size, and orientation of the input texture even as they move according to the specified flow. We cast this problem into an optimization framework that tries to simultaneously satisfy the two (potentially competing) objectives of similarity to the input texture and consistency with the flow field. This optimization is a simple extension of the approach used for unconstrained texture synthesis.

A general framework for example-based synthesis and rendering is also presented. This framework provides a *design space* for constructing example-based rendering algorithms. The goal of such algorithms would be to use texture exemplars to render animations for

which certain behavioral characteristics need to be controlled. Our motion-controlled texture synthesis technique is an instantiation of this framework where the characteristic being controlled is motion represented as a flow field.

CHAPTER I

INTRODUCTION

Textural phenomena are ubiquitous in our environment. Examples include static textures such as reptile skin, wood grain, brick wall, keys on a keyboard, surface of bread, etc, and dynamic textures such as ocean waves, waterfall, smoke from a chimney, fire burning in a fireplace, grass fields waving in the wind, etc. According to the Free Online Dictionary of Computing (FOLDOC) [30], texture is defined as "a measure of the variation of the intensity of a surface, quantifying properties such as smoothness, coarseness and regularity". A common characteristic among textural phenomena is that their appearance is statistically self-similar over space and/or time. One of the long standing goals of computer graphics is to achieve photo-realism in the animation and rendering of such phenomena.

This dissertation presents techniques for realistic synthesis of both static and dynamic textural phenomena using *exemplars* – sample images and videos from the real world. Specifically, we consider the following problems:

Unconstrained Texture Synthesis: Given a sample image, we want to extend it spatially such that the appearance of the synthesized image is texturally similar to input. In other words, properties like shape, size, orientation, etc of *elements* that compose the input texture should be maintained in the output. The same problem is also considered in the case of video texture. For video textures, we want to maintain, not only the spatial appearance of texture elements, but also their dynamic appearance as they evolve over time.

Animating Image Textures using Flow: Given a flow field and an image texture as input, we want to synthesize an animation that has the same perceived motion as that represented by the flow field. However, at the same time, we want to maintain the structure

of texture elements in the synthesized animation. We consider only two-dimensional (2D) flows specified as vector fields that may be time-varying, *i.e.*, dynamically changing over time.

Animating Video Textures using Flow: Here, we want to extend the notion of flow-guided texture animation to use video textures as input. In general, for a textural video, the change in appearance is a combination of flow as well as texture evolution. We want the synthesized animation to convey the motion described by the input flow-field but evolve the texture according to input video. This thesis only partially addresses this problem.

We also present a general framework for controllable synthesis, of which flow-guided synthesis can be considered an instantiation. This framework can be thought of as a design space for addition of constraints and control parameters to the synthesis process.

1.1 Example-based Synthesis

Traditionally, the computer graphics community has relied on simulation of physical processes to achieve realism. For example, global illumination methods that simulate light transport in the scene have been researched very thoroughly and are very mature. Even then, achieving photo-realism for the phenomena mentioned here is a difficult task. This is so because besides simulating light transport, we also need to accurately model the behavior and appearance of the material being rendered. Moreover, in the case of dynamic phenomena, the behavior keeps changing over space and time, adding to the complexity of the rendering process. In recent times, researchers have experimented with the use of real-world images for rendering novel views of a static scene in place of purely simulation based rendering. By using actual samples of the real world, these techniques alleviate some problems associated with modeling of the real world.

Example-based techniques that are specifically concerned with textural phenomena

INPUT

OUTPUT

Figure 1: Example of video texture synthesis. The input river sequence has been extended spatially as well as temporally in the output. Shown are single frames from the input and output video sequences. This result was synthesized using our texture synthesis technique based on graph cuts (see Chapter 3 for more details).

usually exploit the Markov Random Field (MRF) property of textures. This property requires that *locality* and *stationarity* be satisfied. Locality implies that the color at a pixel's location is dependent only on a neighborhood of pixels around it, and stationarity implies that this dependency is independent of the actual location of the pixel. The MRF property has been extensively used for analysis and synthesis of image textures in both graphics and vision communities. The primary objective in image texture synthesis is to extend a given input texture sample spatially; the extension being such that the input sample appears to be a fragment of the extended texture. To some extent, researchers have been successful in adapting these techniques to the video domain as well. In the case of video, the goal is to extend the video not only spatially but also temporally, or to transform it into a continuous loop that can be played infinitely. Most techniques for image texture synthesis cannot be trivially extended to video because of the increase in dimensionality, and at times, due to the lack of isotropy and stationarity in the temporal dimension. In this thesis, we present a framework for synthesizing both image and video textures in a unified fashion using MRF-based optimization techniques that scale well with increase in dimensionality. Figure 1 shows an example of spatio-temporal extension of video achieved using our texture

synthesis technique based on graph cuts (explained in Chapter 3).

1.2 Controllable Texture Synthesis

In order to make texture synthesis more effective, it is imperative to make it more controllable. This is especially true if we want to use it as an alternative to purely synthetic rendering of general scenes. For example, we may want to use a sample image or video of a waterfall to render a synthetic waterfall for which the flow has been computed through physical simulation. The shape and flow of the waterfall in the sample may be significantly different from that of the synthetic one, *e.g.*, the synthetic waterfall may have an obstacle in its path, where the input sample had none. A synthesis technique that only looks at the output size will fail to match the characteristics of the synthetic waterfall. In this scenario, the physical simulation can be thought of as providing a control mechanism for the rendering of the waterfall: we still want a real looking waterfall similar to the input sample; however, we want it to be constrained to match the characteristics output by the simulation.

We present such a synthesis technique that not only maintains visual similarity between the synthesized output and input sample, but also provides a mechanism for controlling certain characteristics of the synthesized output. Our formulation for controllable texture synthesis is cast in the context of rendering dynamic scenes using exemplars. In a way, this formulation is a generalization of classical image-based rendering. In image-based rendering, the objective is to synthesize novel views of a static scene, given images of the scene from certain other viewpoints. We generalize this notion by allowing the scene to be dynamic, as well as by providing a framework for editing scene properties other than just the viewpoint. However, we are also specializing image-based rendering by only allowing scenes that are textural in nature.

We demonstrate our controllable texture synthesis approach by synthesizing dynamic scenes where the motion is controlled using flow-fields, while the appearance is rendered using example textures. Flow is specified as a two-dimensional vector-field (or a sequence

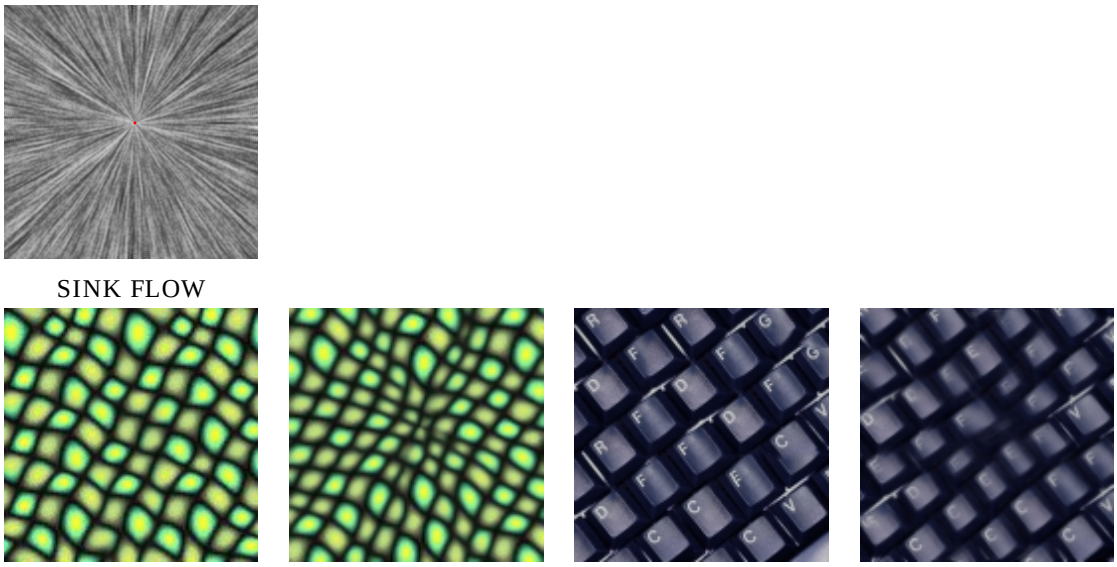


Figure 2: Animating texture using a flow field. Shown are keyframes from texture sequences that follow the sink flow field shown on the top (see Chapter 5 and Chapter 6 for more details).

of it) that may be dynamically changing over time. We have experimented with both image and video textures as input exemplars. Figure 2 shows examples of image textures being animated using a sink flow field (Chapter 5 and Chapter 6 describe the technique used to synthesize these results).

1.3 Contributions

The contributions of this thesis consist of (i) a set of robust techniques for solving the texture synthesis problem for both images and videos, (ii) formulation of a probabilistic (optimization-based) framework for example-based rendering of textural phenomena, and (iii) techniques for animating textures using flow-fields, which demonstrate the power of the synthesis framework formulated in (ii). Following is a brief description of upcoming chapters and how they relate to these contributions.

1.3.1 Image and Video Synthesis using Graph Cuts

This chapter describes a novel patch-based technique for image and video texture synthesis. The output texture is synthesized by copying patches of arbitrary shape and size from the input to the output. There are two key steps in the synthesis process that are repeated iteratively – patch placement and seam optimization. Patch placement finds an appropriate location for the new patch by searching for regions in the input that match well the current state of the output texture. Our search procedure looks for long range interactions between pixels. This facilitates good alignment among the various patches constituting the output texture. Once the location for the new patch is ascertained, seam optimization determines the best portion (arbitrary cut-out) of this new patch that should be copied to the output. This procedure uses a local color-based similarity metric to determine the cost of the seam (boundary of cut-out region that is copied over). The optimization itself is done by embedding the texture into a graph for which the *mincut* corresponds to the minimum-cost seam. This embedding does not impose any constraint on the dimensionality of the texture, and therefore both image and video textures can be synthesized within the same framework.

1.3.2 Framework for Rendering Animations using Texture Exemplars

The synthesis technique based on graph cuts described above is able to generate very realistic looking textures. However, there is limited control over the synthesis process. This chapter describes a general framework for designing rendering techniques that borrow appearance information from example textures, but at the same time allow control over certain characteristics of the output, *e.g.*, motion and shape.

We are interested in rendering dynamic scenes (animations) for which these characteristics can be expressed locally. We define a probabilistic formulation for solving such problems, where the goal is to estimate the maximum *a posteriori* (MAP) appearance of the output. The source (input) texture exemplar serves as a *prior* on the appearance of the

target (output). This prior is based on the MRF assumption described earlier. The consistency of the synthesized target with its desirable characteristics is expressed as a *likelihood* function. We only give a general description of the likelihood function in this chapter, as one needs to design a special likelihood function for each specific characteristic that is being controlled.

1.3.3 Texture Optimization for Unconstrained Synthesis

This chapter builds off from the probabilistic formulation described in the previous chapter. However, here we focus on a novel texture synthesis algorithm that is only based on the prior term described above. The prior term models the similarity of the synthesized target to the source exemplar. Hence, on removal of the likelihood term, the MAP estimate of target appearance corresponds to computing a target texture that is optimally similar to the source texture. In this chapter, we describe a prior model based on a specific texture similarity metric, as well as an algorithm to optimize for it. The similarity metric measures distances between local neighborhoods of the source and target. We optimize for it using an Expectation-Maximization (EM)-like algorithm that iteratively decreases the distance between these local neighborhoods. Note that, in general, it is *NP-hard* to compute the globally optimum target texture corresponding to our prior. Consequently, our optimization technique is designed to only find a locally optimum texture. As with the algorithm based on graph cuts, we can handle both image and video textures with this approach too.

1.3.4 Controllable Synthesis - Flowing Image Textures

In this chapter, we incorporate the characteristic likelihood term of our probabilistic formulation into the optimization-based texture synthesis algorithm mentioned above. We first describe this augmented controllable synthesis algorithm for general likelihood functions. We consider likelihoods that can be expressed as an optimizable energy function. This energy function should measure the cost of the synthesized target with respect to desirable target characteristics. During each iteration of the EM algorithm mentioned above, we

optimize for the likelihood energy in addition to the texture similarity metric.

In the same chapter, we also consider the specific scenario where flow is the target characteristic being controlled while the input exemplar is an image texture. In this case, the likelihood is encoded as the distance (in color-space) between pixels in adjoining frames (of the target) that are connected via flow-lines. We synthesize the target animation frame-by-frame, where each synthesized frame is warped using the target flow to provide an estimate for the next frame. This estimated frame is then used to define the likelihood measure for the new frame. The synthesized texture sequence moves according to the specified flow-field, but maintains textural appearance similar to the source image, even as it evolves over time. We also discuss how to add obstacles in the path of the flow-field and synthesize textures accordingly.

1.3.5 Flowing Video Textures

This chapter extends the flow-guided synthesis scenario described above to the case when the input exemplar is a video texture. Although the likelihood is similar in essence to the one used for image exemplars, in this case, the entire target video is synthesized at once, *i.e.*, not frame-by-frame. This is because the texture similarity metric for video exemplars compares spatio-temporal neighborhoods that span multiple frames. Hence, it is natural to model the output as a 3D spatio-temporal texture as opposed to a sequence of 2D textures.

Synthesis using video exemplars is a much harder optimization problem than synthesis using image exemplars. This is so because the texture at source moves according to its own flow field. At the same time, it may be evolving over time, *i.e.*, texture elements may be changing even if they are not flowing. Decoupling these two processes is a difficult problem and is not addressed in this thesis. Instead, to achieve source-like texture dynamics at the target, we search for source regions with similar flow as that desired at the target on the fly during synthesis. The quality of synthesis is then limited by the degree of mismatch between source and target flow. We analyze this limitation in more detail and explore

possible future research directions to improve synthesis quality for video exemplars, at the end of this chapter.

CHAPTER II

BACKGROUND

Our research is related most closely to image-based rendering and texture synthesis. It aims at generalizing image-based rendering by allowing modification of scene properties other than just the viewpoint. The mathematical and computational tools employed in the process borrow from as well as add to the texture synthesis literature.

2.1 Image-based Rendering

Image-based rendering is primarily concerned with the synthesis of novel views of a static scene, given a set of existing views. The sophistication and complexity of a particular image-based rendering technique generally depends on the freedom it affords in the choice of viewing configurations. The simplest case is when all images share the same viewpoint, but differ in viewing angles. Novel view synthesis then amounts to creation of panoramic mosaics [10, 60, 56], *i.e.*, alignment and stitching of the given images. In the most general case, one can reconstruct the entire 5-dimensional plenoptic function [41], which determines the distribution of light at any location and orientation in space. Researchers have experimented with projections of this function into lower dimensional subspaces [24, 36, 53, 55] – these subspaces correspond to various kinds of constraints on the camera locations and orientations used for obtaining source images or synthesizing novel ones.

The limitation of static scenes in image-based rendering is also its strength. It allows for a physically sound treatment of the problem, leading to solutions that can be easily tested for correctness. Our interest lies in synthesizing videos of phenomena that may be dynamic in nature. The editing capabilities we desire not only include novel viewpoints but

also novel scene characteristics. This requires us to make additional assumptions about the scene: we have chosen to deal with phenomena that can be visually described as texture.

2.2 Texture Synthesis

The objective in texture synthesis is to generate a newer form of output from a smaller example. The need for such a capability is widely recognized to be important for computer graphics applications. For example, sample-based image texture synthesis methods are needed to generate large realistic textures for rendering of complex graphics scenes. Texture is usually defined as an infinite pattern that can be modeled by a stationary stochastic process. For synthesizing texture from a given example, one needs to mimic the underlying stochastic process responsible for generating the example. These arguments hold equally well for both spatial (image) and temporal (video) textures. However, in the case of video textures, the stochastic process in the temporal dimension is generally separate from the process in the spatial dimensions.

Texture synthesis techniques that generate an output texture from an example input can be roughly categorized into three classes. The first class uses a fixed number of parameters within a compact parametric model to describe a variety of textures. Heeger and Bergen [28] use color histograms across frequency bands as a texture description. Portilla and Simoncelli's model [46] includes a variety of wavelet features and their relationships, and is probably the best parametric model for image textures to date. Szummer and Picard [61], Soatto et al. [57], and Wang and Zhu [63] have proposed parametric representations for video. Brand [5] uses low-dimensional subspaces to model high-dimensional video sequences. Parametric models cannot synthesize as large a variety of textures as other models described here, but provide better model generalization and are more amenable to introspection and recognition [47]. They therefore perform well for analysis of textures and can provide a better understanding of the perceptual process.

The second class of texture synthesis methods is non-parametric, which means that

rather than having a fixed number of parameters, they use a collection of *exemplars* to model the texture. DeBonet [14], who pioneered this group of techniques, samples from a collection of multi-scale filter responses to generate textures. Efros and Leung [17] were the first to use an even simpler approach, directly generating textures by copying pixels from the input texture. Wei and Levoy [64] extended this approach to multiple frequency bands and used vector quantization to speed up the processing. BarJoseph et al. [2] invented a pixel-based technique for synthesis of video. All these techniques generate textures one pixel at a time.

The third, most recent class of techniques generates textures by copying whole *patches* from the input. Ashikmin [1] made an intermediate step towards copying patches by using a pixel-based technique that favors transfer of coherent patches. Liang et al. [38], Efros and Freeman [18], and Guo et al. [26] explicitly copy whole patches of input texture at a time. Cohen et al. [11] pre-compute a set of image patches called Wang Tiles, which they then use for synthesizing textures. Wu and Yu [67] combine feature-guided search with patch-based synthesis. Schödl et al. [51] perform video synthesis by copying whole frames from the input sequence. This last class of techniques arguably creates the best synthesis results on the largest variety of textures. These methods, unlike the parametric methods described above, yield a limited amount of information for texture analysis.

The texture synthesis techniques presented in this thesis are most naturally classified as patch-based techniques. We present two techniques for unconstrained texture synthesis. Our first technique, based on graph cuts, stitches large patches of arbitrary sizes along perceptually optimal seams. On the other hand, our texture optimization technique considers patch sizes that vary from large to small and attempts to decrease the mismatch between overlapping patches. This technique is really intermediate between patch and pixel-based techniques; however, the primary structure of the synthesized texture is still determined by the larger patches. This variability in the patch size also allows us to extend the technique to handle controllable synthesis.

2.3 Markov Random Fields

Across different synthesis techniques, textures are often described as Markov Random Fields (MRFs) [14, 43, 17, 18, 64]. This interpretation allows texture to be analyzed or synthesized by only considering interactions between neighboring pixels. Analysis and optimization in MRFs have been studied extensively in the context of computer vision [37]. Within the context of MRFs, researchers have also investigated multi-resolution [9, 8, 14, 64] and cluster-based [45, 39, 64, 71] approaches for representing textures. These have proven to be useful as compact descriptions of textures, besides aiding in the development of efficient sampling and optimization techniques for texture analysis and synthesis.

Our techniques for texture synthesis also formulate texture as a Markov Random Field. For our patch stitching technique, we use a measure for seam quality that is defined in terms of pairs of pixels. This is a standard way of representing costs in an MRF; we use a graph cut technique to optimize the likelihood of the MRF. Among other techniques using graph cuts [25], we have chosen a technique by Boykov et al. [4], which is particularly suited for the type of cost function found in texture synthesis.

Note that the graph cut algorithm uses the MRF formulation to define only the seam cost, which is used to paste a single patch over an existing synthesized output. In our texture optimization technique, however, we have extended the MRF formulation to define a cost for the entire texture as a whole. This cost measures the quality of the synthesized texture with respect to the input texture. We have developed a novel technique for MRF optimization in this context using the Expectation-Maximization (EM) [40] algorithm. There also exist other techniques that use MRF optimization for texture synthesis. In particular, Paget and Longstaff [43] have used local annealing over a multi-scale MRF for texture synthesis. However, they consider only pixel-to-pixel interactions. In contrast, our technique can handle interactions between large neighborhoods by virtue of being patch-based. Also, our EM-based optimization is faster than their annealing-based approach, but still gives high

quality results. Freeman et al. [23] have used belief propagation over an MRF for super-resolution. They can handle interactions across large neighborhoods. However, theirs is a fully discrete optimization while our approach is semi-discrete-continuous, which leads to a simpler optimization algorithm. Additionally, this also implies that in our approach, all synthesized pixels are not necessarily *copied* from the input, which makes the technique more flexible and accommodating for constrained synthesis.

In image analysis, Jojic et al. [32] use a distance metric similar to our texture energy metric for computing image epitomes. Fitzgibbon et al. [21] also use a similar metric as texture prior for image-based rendering, while Wexler et al. [66] use it for hole-filling in video. Wei and Levoy [65] use a global pixel-based approach for texture synthesis where non-causal neighborhoods are used to simultaneously determine each pixel of the evolving texture; we, on the other hand, directly combine these neighborhoods as patches within our optimization.

2.4 Controllable Texture Synthesis

Although most work in texture synthesis has been focussed on unconstrained synthesis, there has been some research for adding higher-level user control to these methods. Ashikhmin [1], requires a user to specify the large scale properties of the output texture through a painting-like interface. Efros and Freeman [18] perform texture transfer on arbitrary images by matching correspondence maps. Hertzmann et al. [29] synthesize the output of a filter or painting operation on a given image by learning the operation from a prior input-output image pair. Brooks and Dodgson [7] perform texture editing by replicating local editing operations globally over the texture using self-similarity. Tonietto and Walter [62] allow local scale variations by blending textures at different resolutions appropriately. Zhang et al. [70] control features like scale, orientation, and shape of the synthesized texture by explicitly modeling texture elements. All of the above methods support some sort of a higher-level control for spatial synthesis.

For video textures, Schödl et al. [51] blend multiple videos at varying speeds to control the speed of the synthesized video. They also use video sprites for synthesizing character animations from video [51, 49, 50]. Doretto and Soatto [16] edit the speed, intensity, and scale of video textures by editing parameters of a Linear Dynamical System (LDS). Yuan et al. [68] use a closed-loop linear dynamical system to synthesize dynamic textures. Bregler et al. [6] and Ezzat et al. [20] use speech to control facial animation synthesis from video data. Sun et al. [59] extract wind speed and harmonic oscillator parameters to control the motion of synthetic objects in a real natural scene. All of the above are examples of higher level control (*i.e.*, specified trajectories, matching to data, or speech) applied to temporal synthesis.

Bhat et al. [3] provide an interactive system for editing video by specifying flow lines in the input and output sequences. Unlike other efforts on controlled video synthesis, the goal of this work is similar to that of ours as in low-level control is imparted to synthesize new video. However, our controllable synthesis algorithms differ completely. Our synthesis technique can also be considered more general as it can handle arbitrary time-varying flow-fields, and can also use image textures as input. Additionally, we provide a general framework for controllable texture synthesis that can be specialized to control other characteristics besides flow.

CHAPTER III

IMAGE AND VIDEO TEXTURE SYNTHESIS USING GRAPH CUTS

In this chapter, we present a new method for texture synthesis in both the image and the video domain[1]. Our technique is example-based; using a small example patch of the texture, we generate a larger pattern with similar stochastic properties. Specifically, our approach for texture synthesis generates textures by copying input texture patches. Our algorithm first searches for an appropriate location to place the patch; it then uses a *graph cut* technique to find the optimal region of the patch to transfer to the output. In our approach, textures are not limited to spatial (image) textures, and include spatio-temporal (video) textures. In addition, our algorithm supports iterative refinement of the output by allowing for successive improvement of the patch seams.

When synthesizing a texture, we want the generated texture to be perceptually similar to the example texture. This concept of perceptual similarity has been formalized as a Markov Random Field (MRF). The output texture is represented as a grid of nodes, where each node refers to a pixel or a neighborhood of pixels in the input texture. The marginal probability of a pair of nodes depends on the similarity of their pixel neighborhoods, so that pixels from similar-looking neighborhoods in the input texture end up as neighbors in the generated texture, preserving the perceptual quality of the input. The goal of texture synthesis can then be restated as the solution for the nodes of the network, that maximizes the total likelihood. This formulation is well-known in machine-learning as the problem of

[1]This work was done in collaboration with Arno Schödl and a related joint publication is [35]. This chapter presents all parts of the associated research for completeness. Please also refer to Arno Schödl's thesis [48] for exhaustive review.

probabilistic inference in graphical models and is proven to be *NP-hard* in case of cyclic networks. Hence, all techniques that model the texture as a MRF [14, 17, 18, 64] compute some approximation to the optimal solution.

In particular, texture synthesis algorithms that generate their output by copying patches (or their generalizations to higher dimensions) must make two decisions for each patch: (1) where to position the input texture relative to the output texture (the *offset* of the patch), and (2) which parts of the input texture to transfer into the output space (the patch *seam*) (Figure 3). The primary contribution of this research is an algorithm for texture synthesis, which after finding a good patch offset, computes the best patch seam (the seam yielding the highest possible MRF likelihood among all possible seams for that offset). The algorithm works by reformulating the problem as a minimum cost graph cut problem: the MRF grid is augmented with special nodes, and a minimum cut of this grid between two special terminal nodes is computed. This minimum cut encodes the optimal solution for the patch seam. We also propose a set of algorithms to search for the patch offset at each iteration. These algorithms try to maintain the large scale structure of the texture by matching large input patches with the output. An important observation is that the flexibility of the our seam optimization technique to paste large patches at each iteration in a non-causal fashion is really what permits the design of our offset search algorithms. The offset searching and seam finding methods are therefore complementary to each other, and work in tandem to generate the obtained results.

Efros and Freeman [18] were the first to incorporate seam finding by using dynamic programming. However, dynamic programming imposes an artificial grid structure on the pixels and therefore does not treat each pixel uniformly. This can potentially mean missing out on good seams that cannot be modeled within the imposed structure. Moreover, dynamic programming is a memoryless optimization procedure and cannot explicitly improve existing seams. This restricts its use to appending new patches to existing textures.

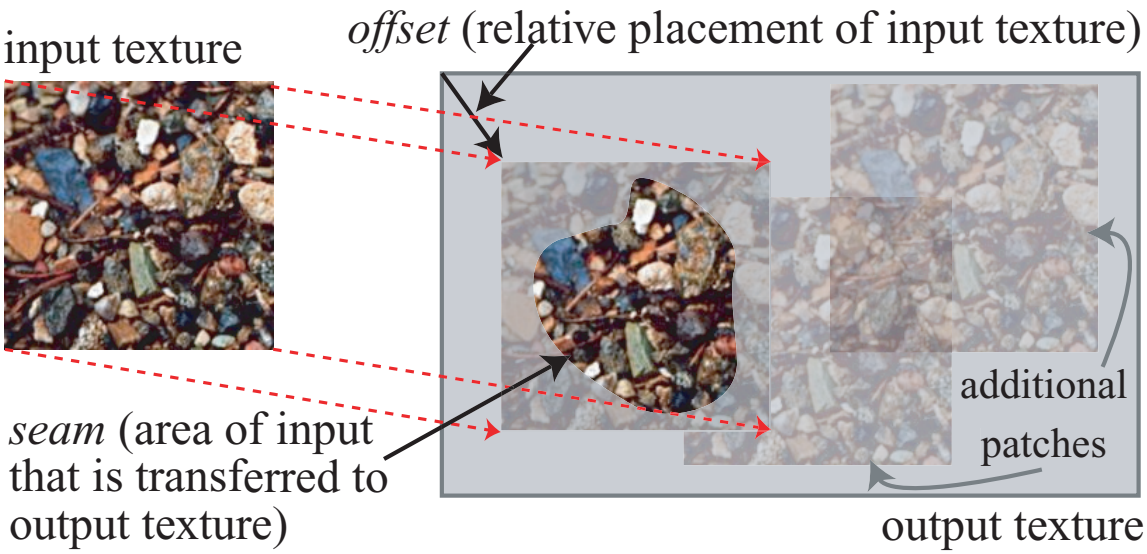


Figure 3: Image texture synthesis by placing small patches at various offsets followed by the computation of a seam that enforces visual smoothness between the existing pixels and the newly placed patch.

Our graph cut method treats each pixel uniformly and is also able to place patches *over* existing texture. Unlike dynamic programming, which is restricted to 2D, the seam optimization presented here generalizes to any dimensionality. Based on this seam optimization, we have developed algorithms for both two and three dimensions to generate spatial (2D, images) and spatio-temporal (3D, video) textures.

Finally, we have extended our algorithm to allow for multiple scales and different orientations which permits the generation of larger images with more variety and perspective variations. We have also implemented an interactive system that allows for merging and blending of different types of images to generate composites without the need for any a *priori* segmentation.

3.1 Patch Fitting using Graph Cuts

We synthesize new texture by copying irregularly shaped patches from the sample image into the output image. The patch copying process is performed in two stages. First a candidate rectangular patch (or patch offset) is selected by performing a comparison between

the candidate patch and the pixels already in the output image. We describe our method of selecting candidate patches in a later section (Section 3.2). Second, an optimal (irregularly shaped) portion of this rectangle is computed and only these pixels are copied to the output image (Figure 3). This process is repeated iteratively so as to improve the perceptual quality of the synthesized texture, as shown in Figure 4. The portion of the patch to copy is determined by using a graph cut algorithm, and this is the heart of our synthesis technique.

In order to introduce the graph cut technique, we first describe how it can be used to perform texture synthesis in the manner of Efros and Freeman's image quilting [18]. Later we will see that it is a much more general tool. In image quilting, small blocks (*e.g.,* 32×32 pixels) from the sample image are copied to the output image. The first block is copied at random, and then subsequent blocks are placed such that they partly overlap with previously placed blocks of pixels. The overlap between old and new blocks is typically 4 or 8 pixels in width. Efros and Freeman use dynamic programming to choose the minimum cost path from one end of this overlap region to the other. That is, the chosen path is through those pixels where the old and new patch colors are similar (Figure 5(left)). The path determines which patch contributes pixels at different locations in the overlap region.

To see how this can be cast into a graph cut problem, we first need to choose a matching quality measure for pixels from the old and new patch. In the graph cut version of this problem, the selected path will run *between* pairs of pixels. The simplest quality measure, then, will be a measure of color difference between the pairs of pixels. Let s and t be two adjacent pixel positions in the overlap region. Also, let $\mathbf{A}(s)$ and $\mathbf{B}(s)$ be the pixel colors at the position s in the old and new patches, respectively. We define the matching quality cost M between the two adjacent pixels s and t that copy from patches $\mathbf{A}$ and $\mathbf{B}$ respectively to be:

$$M(s,t,\mathbf{A},\mathbf{B}) = \|\mathbf{A}(s) - \mathbf{B}(s)\| + \|\mathbf{A}(t) - \mathbf{B}(t)\| \tag{1}$$

where $\|\cdot\|$ denotes an appropriate norm. We consider a more sophisticated cost function in

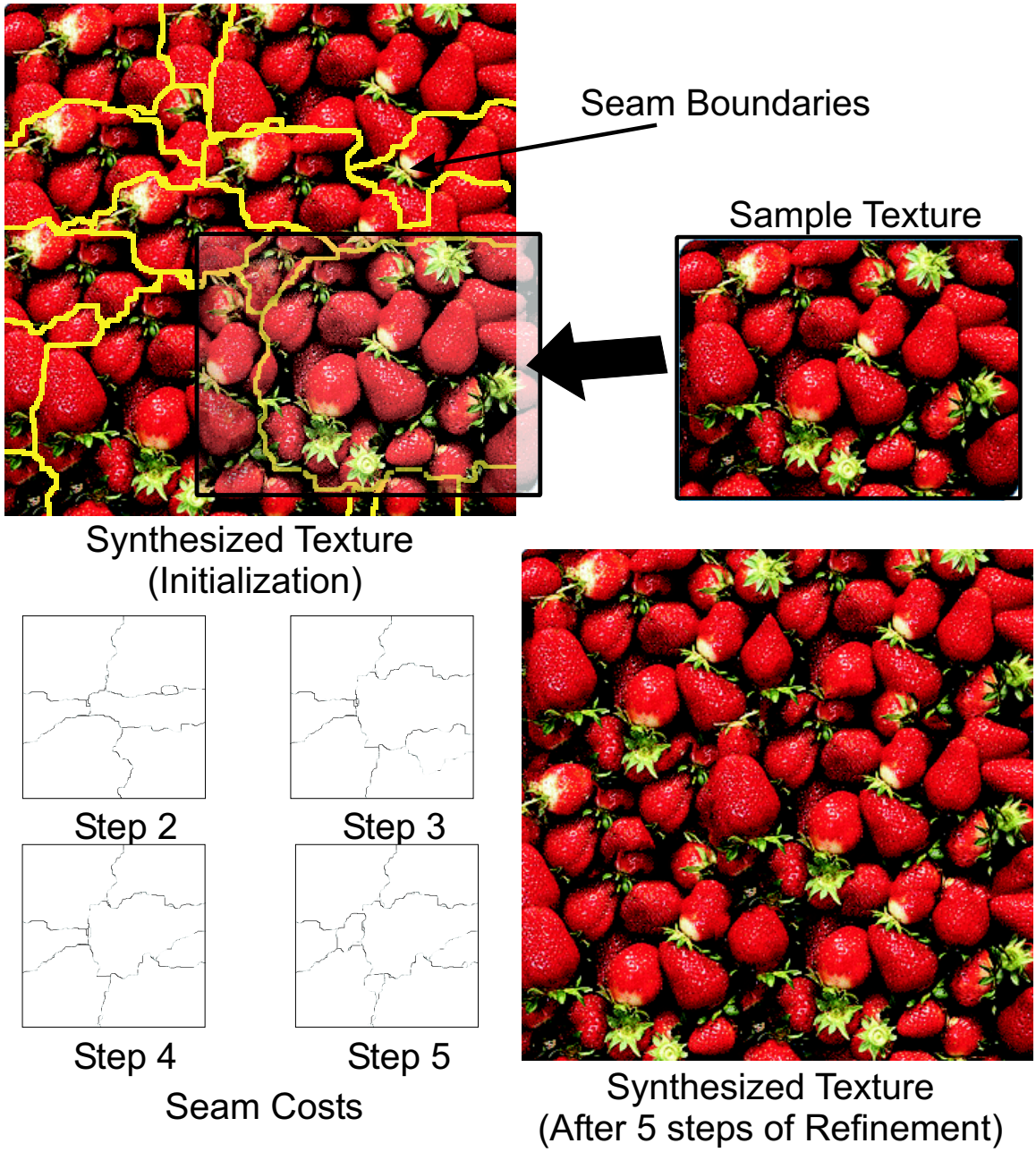


Figure 4: This figure illustrates the process of synthesizing a larger texture from an example input texture. Once the texture is initialized, we find new patch locations appropriately so as to refine the texture. Note the irregular patches and seams. Seam error measures that are used to guide the patch selection process are shown. This process is also shown in the video.

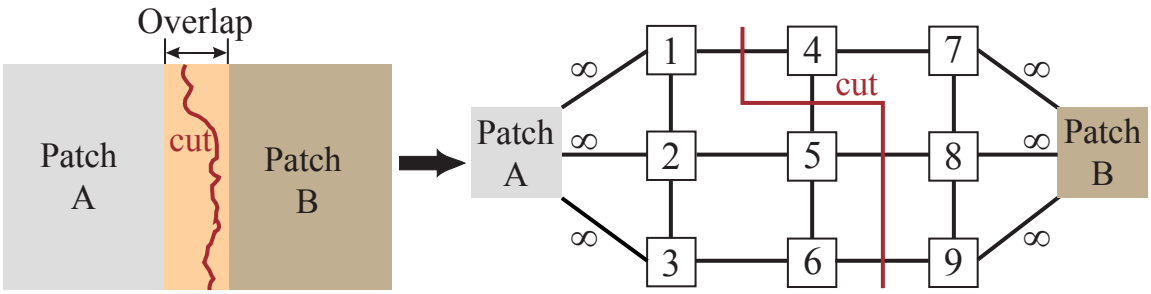


Figure 5: (Left) Schematic showing the overlapping region between two patches. (Right) Graph formulation of the seam finding problem, with the red line showing the minimum cost cut.

a later section. For now, this match cost is all we need to use graph cuts to solve the path finding problem.

Consider the graph shown in Figure 5(right) that has one node per pixel in the overlap region between patches. We wish to find a low-cost path through this region from top to bottom. This region is shown as 3×3 pixels in the figure, but it is usually more like 8×32 pixels in typical image quilting problems (the overlap between two 32×32 patches). The arcs connecting the adjacent pixel nodes s and t are labelled with the matching quality cost $M(s,t,\mathbf{A},\mathbf{B})$. Two additional nodes are added, representing the old and new patches (**A** and **B**). Finally, we add arcs that have infinitely high costs between some of the pixels and the nodes **A** or **B**. These are *constraint* arcs, and they indicate pixels that we insist will come from one particular patch. In Figure 5, we have constrained pixels 1, 2, and 3 to come from the old patch, and pixels 7, 8, and 9 must come from the new patch. To find out which patch each of the pixels 4, 5, and 6 will come from is determined by solving a graph cut problem. Specifically, we seek the minimum cost cut of the graph, that separates node **A** from node **B**. This is a classical graph problem called min-cut or max-flow [22, 52] and algorithms for solving it are well understood and easy to code. In the example of Figure 5, the red line shows the minimum cut, and this means pixel 4 will be copied from patch **B** (since its portion of the graph is still connected to node **B**), whereas pixels 5 and 6 will be from the old patch **A**.

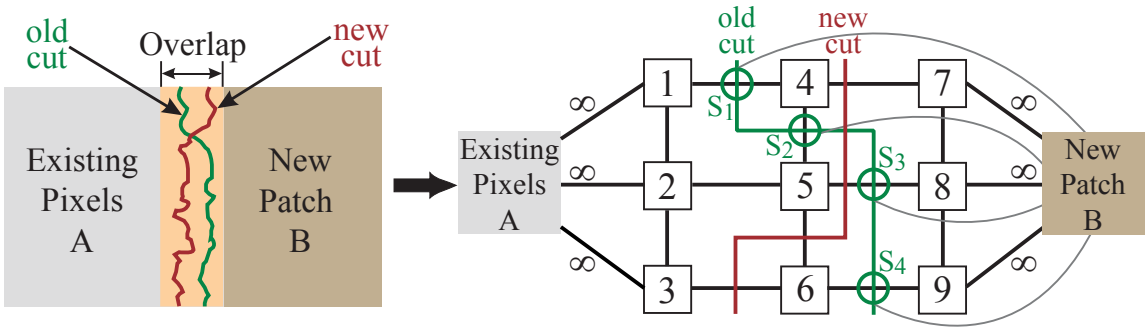


Figure 6: (Left) Finding the best new cut (red) with an old seam (green) already present. (Right) Graph formulation with old seams present. Nodes s_1 to s_4 and their arcs to **B** encode the cost of the old seam.

3.1.1 Accounting for Old Seams

The above example does not show the full power of using graph cuts for texture synthesis. Suppose that several patches have already been placed down in the output texture, and that we wish to lay down a new patch in a region where multiple patches already meet. There is a potential for visible seams along the border between old patches, and we can measure this using the arc costs from the graph cut problem that we solved when laying down these patches. We can incorporate these old seam costs into the new graph cut problem, and thus we can determine which pixels (if any) from the new patch should cover over some of these old seams. To our knowledge, this cannot be done using dynamic programming – the old seam and its cost at each pixel needs to be *remembered*; however, dynamic programming is a memoryless optimization procedure in the sense that it cannot keep track of old solutions.

We illustrate this problem in Figure 6. In the graph formulation of this problem, all of the old patches are represented by a single node **A**, and the new patch is **B**. Since **A** now represents a collection of patches, we use $\mathbf{A}_s$ to denote the particular patch that pixel s copies from. For each seam between old pixels, we introduce a *seam node* into the graph between the pair of pixel nodes. We connect each seam node with an arc to the new patch node **B**, and the cost of this arc is the old matching cost when we created this seam, *i.e.*, $M(s,t,\mathbf{A}_s,\mathbf{A}_t)$ where s and t are the two pixels that straddle the seam. In Figure 6, there is

an old seam between pixels 1 and 4, so we insert a seam node s_1 between these two pixel nodes. We also connect s_1 to the new patch node **B**, and label this arc with the old matching cost $M(1,4,\mathbf{A}_1,\mathbf{A}_4)$. We label the arc from pixel node 1 to s_1 with the cost $M(1,4,\mathbf{A}_1,\mathbf{B})$ (the matching cost when only pixel 4 is assigned the new patch) and the arc from s_1 to pixel node 4 with the cost $M(1,4,\mathbf{B},\mathbf{A}_4)$ (the matching cost when only pixel 1 is assigned the new patch). If the arc between a seam node and the new patch node **B** is cut, this means that the old seam remains in the output image. If such an arc is *not* cut, this means that the seam has been overwritten by new pixels, so the old seam cost is not counted in the final cost. If one of the arcs between a seam node and the pixels adjacent to it is cut, it means that a new seam has been introduced at the same position and a new seam cost (depending upon which arc has been cut) is added to the final cost. In Figure 6, the red line shows the final graph cut: the old seam at s_3 has been replaced by a new seam, the seam at s_4 has disappeared, and fresh seams have been introduced between nodes 3 and 6, 5 and 6, and 4 and 7.

This equivalence between seam cost and the min-cut of the graph holds if and only if at most one of the three arcs meeting at the seam nodes is included in the min-cut. The cost of this arc is the new seam cost, and if no arc is cut, the seam is removed and the cost goes to zero. This is true only if we ensure that M is a metric (satisfies the triangle inequality) [4], which is true if the norm in (1) is a metric. Satisfying the triangle inequality implies that picking two arcs originating from a seam node is always costlier than picking just one of them, hence at most one arc is picked in the min-cut, as desired. Our graph cut formulation is equivalent to the one in [4] and the addition of patches corresponds to the α-expansion step in their work. In fact, our implementation uses their code for computing the graph min-cut. Whereas they made use of graph cuts for image noise removal and image correspondence for stereo, our use of graph cuts for texture synthesis is novel.

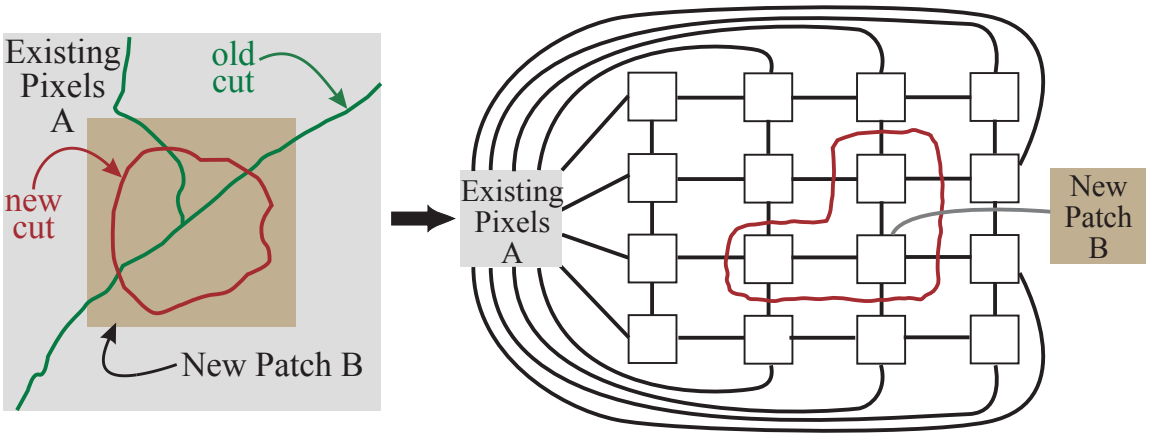


Figure 7: (Left) Placing a patch surrounded by already *filled* pixels. Old seams (green) are partially overwritten by the new patch (bordered in red). (Right) Graph formulation of the problem. Constraint arcs to **A** force the border pixels to come from old image. Seam nodes and their arcs are not shown in this image for clarity.

3.1.2 Surrounded Regions

So far we have shown new patches that overlap old pixels only along a border region. In fact, it is quite common in our synthesis approach to attempt to place a new patch over a region where the entire area has already been covered by pixels from earlier patch placement steps. This is done in order to overwrite potentially visible seams with the new patch, and an example of this is shown in Figure 7. The graph formulation of this problem is really the same as the problem of Figure 6. In this graph cut problem, all of the pixels in a border surrounding the placement region are constrained to come from existing pixels. These constraints are reflected in the arcs from the border pixels to node **A**. We have also placed a single constraint arc from one interior pixel to node **B** in order to force at least one pixel to be copied from patch **B**. In fact, this kind of a constraint arc to patch **B** isn't even required. To avoid clutter in this figure, the nodes and arcs that encode old seam costs have been omitted. These omitted nodes make many connections between the central portion of the graph and node **B**, so even if the arc to **B** were removed, the graph would still be connected. In the example, the red line shows how the resulting graph cut actually forms a closed loop, which defines the best irregularly-shaped region to copy into the output image.

Finding the best cut for a graph can have a worst-case $O(n^2)$ cost for a graph with n nodes [52]. For the kinds of graphs we create, however, we never approach this worst-case behavior. Our timings appear to be $O(n\log(n))$.

3.2 Patch Placement & Matching

Now we describe several algorithms for picking candidate patches. We use one of three different algorithms for patch selection, based on the type of texture we are synthesizing. These selection methods are: (1) *random placement*, (2) *entire patch matching*, and (3) *sub-patch matching*.

In all these algorithms, we restrict the patch selection to previously unused offsets. Also, for the two matching-based algorithms, we first find a region in the current texture that needs a lot of improvement. We use the cost of existing seams to quantify the error in a particular region of the image, and pick the region with the largest error. Once we pick such an *error-region*, we force the patch selection algorithm to pick only those patch locations that completely overlap the error-region. When the texture is being initialized, *i.e.*, when it is not completely covered with patches of input texture, the error-region is picked differently and serves a different purpose: it is picked so that it contains both initialized and uninitialized portions of the output texture – this ensures that the texture is extended by some amount and also that the extended portion is consistent with the already initialized portions of the texture.

Now we discuss the three patch placement and matching methods in some detail. The same three placement algorithms are used for synthesis of image (spatial) and video (spatio-temporal) textures, discussed in Sections 3.4 and 3.5 respectively. Note that patch placement is really just a translation applied to the input before it is added to the output.

Random placement: In this approach, the new patch, (the entire input image), is translated to a random offset location. The graph cut algorithm selects a piece of this patch to

lay down into the output image, and then we repeat the process. This is the fastest synthesis method and gives good results for random textures.

Entire patch matching: This involves searching for translations of the input image that match well with the currently synthesized output. To account for partial overlaps between the input and the output, we normalize the sum-of-squared-differences (SSD) cost with the area of the overlapping region. We compute this cost for all possible translations of the input texture as:

$$C(t) = \frac{1}{|\mathbf{A}_t|} \sum_{p \in \mathbf{A}_t} |\mathbf{I}(p) - \mathbf{O}(p+t)|^2 \tag{2}$$

where $C(t)$ is the cost at translation t of the input, $\mathbf{I}$ and $\mathbf{O}$ are the input and output images respectively, and $\mathbf{A}_t$ is the portion of the translated input overlapping the output. We pick the new patch location stochastically from among the possible translations according to the probability function:

$$P(t) \propto e^{-\frac{C(t)}{k\sigma^2}} \tag{3}$$

where σ is the standard deviation of the pixel values in the input image and k controls the randomness in patch selection. A low value of k leads to picking of only those patch locations that have a very good match with the output whereas a high value of k leads to more random patch selection. In our implementation, we varied k between 0.001 and 1.0. Note that we also constrain the selected patch to overlap the error-region as described above. This method gives the best results for structured and semi-structured textures since their inherent periodicity is captured very well by this cost function.

Sub-patch matching: This is the most general of all our patch selection techniques. It is also the best method for stochastic textures as well as for video sequences involving textural motion such as water waves and smoke (Section 3.5). The motion in such sequences is spatially quite unstructured with different regions of the image exhibiting different motions; however, the motion itself is structured in that it is locally coherent. In sub-patch matching,

we first pick a small sub-patch (which is usually significantly smaller than the input texture) in the output texture. In our implementation, this *output-sub-patch* is the same or slightly larger than the error-region that we want to place the patch over. We now look for a sub-patch in the input texture that matches this output-sub-patch well. Equivalently, we look for translations of the input such that the portion of the input overlapping the output-sub-patch matches it well – only those translations that allow complete overlap of the input with the output-sub-patch are considered. The cost of a translation of the input texture is now defined as:

$$C(t) = \sum_{p \in \mathbf{S_O}} |\mathbf{I}(p-t) - \mathbf{O}(p)|^2 \tag{4}$$

where $\mathbf{S_O}$ is the output-sub-patch and all other quantities are the same as in (2). The patch is again picked stochastically using a probability function similar to (3).

3.3 Extensions & Refinements

Now we briefly describe a few improvements and extensions that we have implemented for image and video synthesis. These extensions include improvements to the cost functions that account for frequency variations, inclusion of feathering and multi-resolution techniques to smooth out visible artifacts, and speed ups in the SSD-based algorithms used in patch matching.

Adapting the Cost Function: The match cost in (1) used in determining the graph cut does not pay any attention to the frequency content present in the image or video. Usually, discontinuities or seam boundaries are more prominent in low frequency regions rather than high frequency ones. We take this into account by computing the gradient of the image or video along each direction – horizontal, vertical and temporal (in case of video) – and scale the match cost in (1) appropriately, resulting in the following new cost function.

$$M'(s,t,\mathbf{A},\mathbf{B}) = \frac{M(s,t,\mathbf{A},\mathbf{B})}{\|\mathbf{G}^d_\mathbf{A}(s)\| + \|\mathbf{G}^d_\mathbf{A}(t)\| + \|\mathbf{G}^d_\mathbf{B}(s)\| + \|\mathbf{G}^d_\mathbf{B}(t)\|} \tag{5}$$

Here, d indicates the direction of the gradient and is the same as the direction of the edge between s and t. $\mathbf{G}_{\mathbf{A}}^{d}$ and $\mathbf{G}_{\mathbf{B}}^{d}$ are the gradients in the patches **A** and **B** along the direction d. M' penalizes seams going through low frequency regions more than those going through high frequency regions, as desired.

Feathering and multi-resolution splining: Although graph cuts produce the best possible seam around a given texture patch, it can still generate visible artifacts when no good seam exists at that point. It is possible to hide these artifacts by feathering the pixel values across seams. For every pixel s close enough to a seam, we find all patches meeting at that seam.

The pixel s is then assigned the weighted sum of pixel values $\mathbf{P}(s)$ corresponding to each such patch **P**. Most of the time, this form of feathering is done using a Gaussian kernel.

We also use multi-resolution splining [8] of patches across seams, which is helpful when the seams are too obvious, but it also tends to reduce the contrast of the image or video when a lot of small patches have been placed in the output. In general, we have found it useful to pick between feathering and multi-resolution splining on a case-by-case basis.

FFT-Based Acceleration: The SSD-based algorithms described in Section 3.2 can be computationally expensive if the search is carried out naively. Computing the cost $C(t)$ for all valid translations is $O(n^2)$ where n is the number of pixels in the image or video. However, the search can be accelerated using *Fast Fourier Transforms (FFT)* [33, 58] . For example, we can rewrite (4) as:

$$C(t) = \sum_{p} \mathbf{I}^2(p-t) + \sum_{p} \mathbf{O}^2(p) - 2\sum_{p} \mathbf{I}(p-t)\mathbf{O}(p) \tag{6}$$

The first two terms in (6) are sum of squares of pixel values over sub-blocks of the image or video and can be computed efficiently in $O(n)$ time using summed-area tables [13]. The

third term is a convolution of the input with the output and can be computed in $O(n\log(n))$ time using FFT. Since n is extremely large for images and especially for video, we get a huge speed up using this method – for a $150 \times 100 \times 30$ video sequence, $n \approx 10^6$, and the time required to search for a new patch reduces from around *10 minutes* (using naive search) to *5 seconds* (using FFT-based search).

3.4 Image Synthesis

We have applied our technique for image and texture synthesis to generate regular, structured and random textures as well as to synthesize extensions of natural images. Figures 8, 9, 10, and 11 show results for a variety of two-dimensional image textures. We used *entire patch matching* as our patch selection algorithm for the TEXT, NUTS, ESCHER, and KEYBOARD images, while *sub-patch matching* was used for generating CHICK PEAS, MACHU PICCHU, CROWDS, SHEEP, OLIVES, BOTTLES, and LILIES. The computation for images is quite fast, mainly due to the use of FFT-based search. All image synthesis results presented here took between 5 seconds and 5 minutes to run. The LILIES image took 5 minutes because it was originally generated to be 1280×1024 in size. We also compare some of our results with that of Image Quilting [18] in Figure 10. Now we briefly describe a few specialized extensions and applications of our 2D texture synthesis technique.

A. Additional Transformations of Source Patches: Our algorithm relies on placing the input patch appropriately and determining a seam that supports efficient patching of input images. Even though we have only discussed the possibility of translating the input patch over the output region, one could generalize this concept to include other *transformations* of the input patch like rotation, scaling, affine or projective. For images, we have experimented with the use of *rotated*, *mirrored*, and *scaled* versions of the input texture. Allowing more transformations gives us more flexibility and variety in terms of the kind of output that can be generated. However, as we increase the potential transformations of the

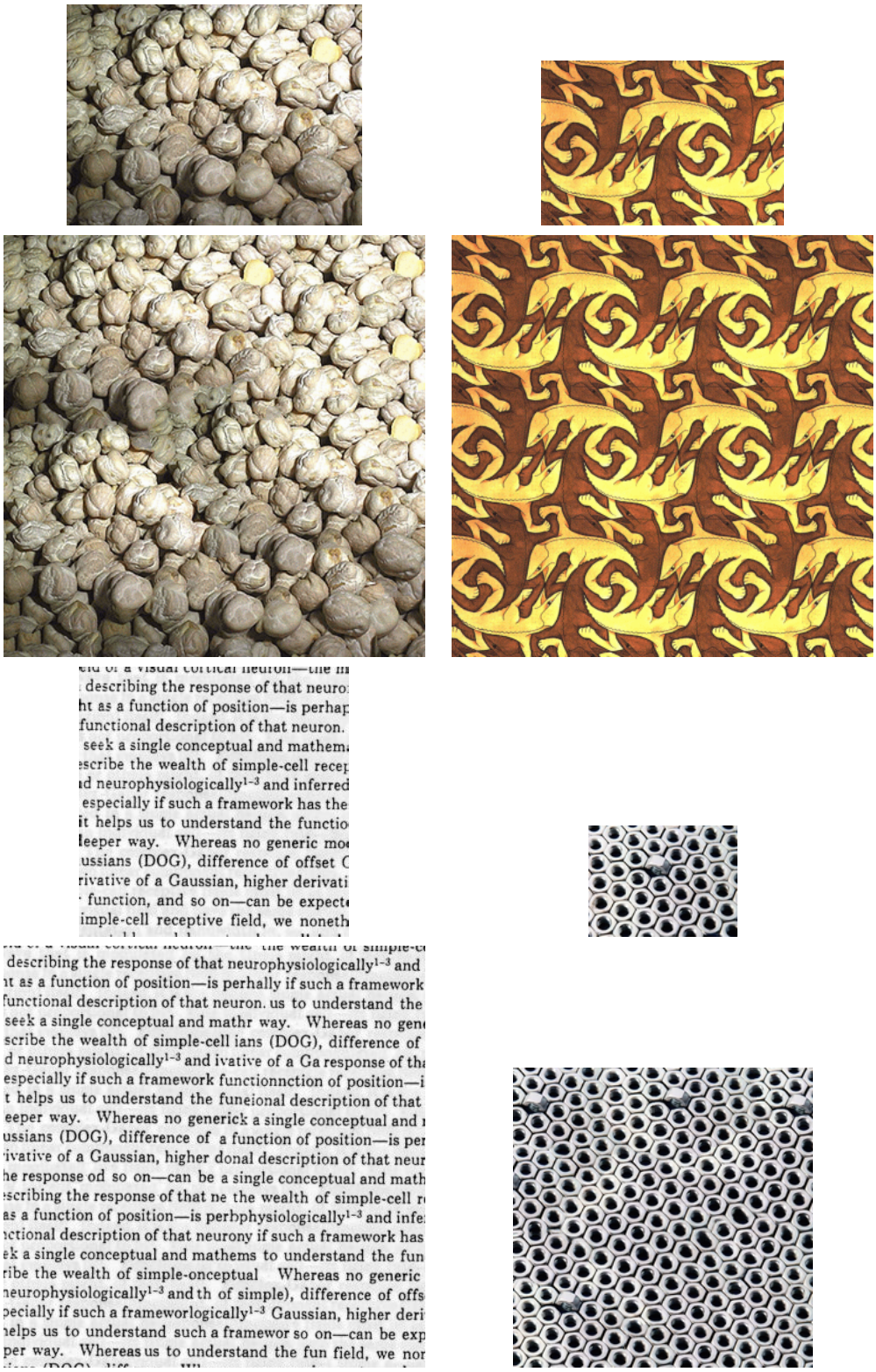


Figure 8: 2D texture synthesis results. The smaller images are the example images used for synthesis. Shown are CHICK PEAS, ESCHER, TEXT, and NUTS from left to right and top to bottom.

Figure 9: 2D synthesis results for natural images. The smaller images are the example images used for synthesis. Shown are MACHU PICCHU©Adam Brostow, CROWDS and SHEEP from top to bottom.

input texture, the cost of searching for good transformations also increases. Therefore, we restrict the transformations other than translations to a small number. Note that the number of candidate translations is of the order of the number of pixels. We generate the transformed versions of the input before we start synthesis. To avoid changing the searching algorithm significantly, we put all the transformed images into a single image juxtaposed against each other. This makes the picking of any transformation equivalent to the picking of a translation. Then, only the portion containing the particular transformed version of the image is sent to the graph cut algorithm instead of the whole mosaic of transformations.

In Figure 10, we make use of rotational and mirroring transformations to reduce repeatability in the synthesis of the OLIVES image. Scaling allows mixing different sizes of texture elements together. One interesting application of scaling is to generate images conveying deep perspective. We can constrain different portions of the output texture to copy from different scales of the input texture. If we force the scale to vary in a monotonic fashion across the output image, it gives the impression of an image depicting perspective. For example, see BOTTLES and LILIES in Figure 11.

B. Interactive Merging and Blending: One application of the graph cut technique is interactive image synthesis along the lines of [42, 7]. In this application, we pick a set of source images and combine them to form a single output image. As explained in the section on patch fitting for texture synthesis (Section 3.1), if we constrain some pixels of the output image to come from a particular patch, then the graph cut algorithm finds the best seam that goes through the remaining unconstrained pixels. The patches in this case are the source images that we want to combine. For merging two such images, the user first specifies the locations of the source images in the output and establishes the constrained pixels interactively. The graph cut algorithm then finds the best seam between the images.

This is a powerful way to combine images that are not similar to each other since the graph cut algorithm finds any regions in the images that *are* similar and tries to make the

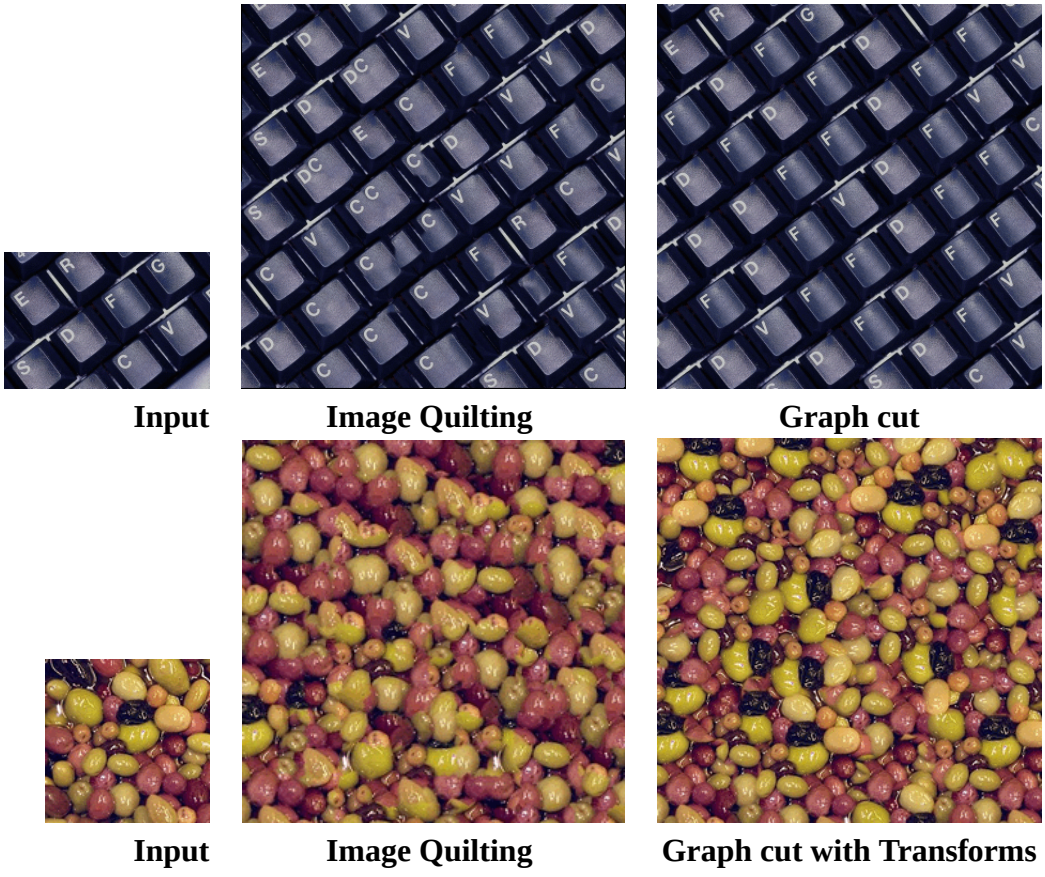


Figure 10: Comparison of our graph cut algorithm with Image Quilting [18]. Shown are KEYBOARD and OLIVES. For OLIVES, rotation and mirroring of patches was used to increase variety in the output. The quilting result for KEYBOARD was generated using our implementation of Image Quilting; the result for OLIVES is courtesy of Efros and Freeman.

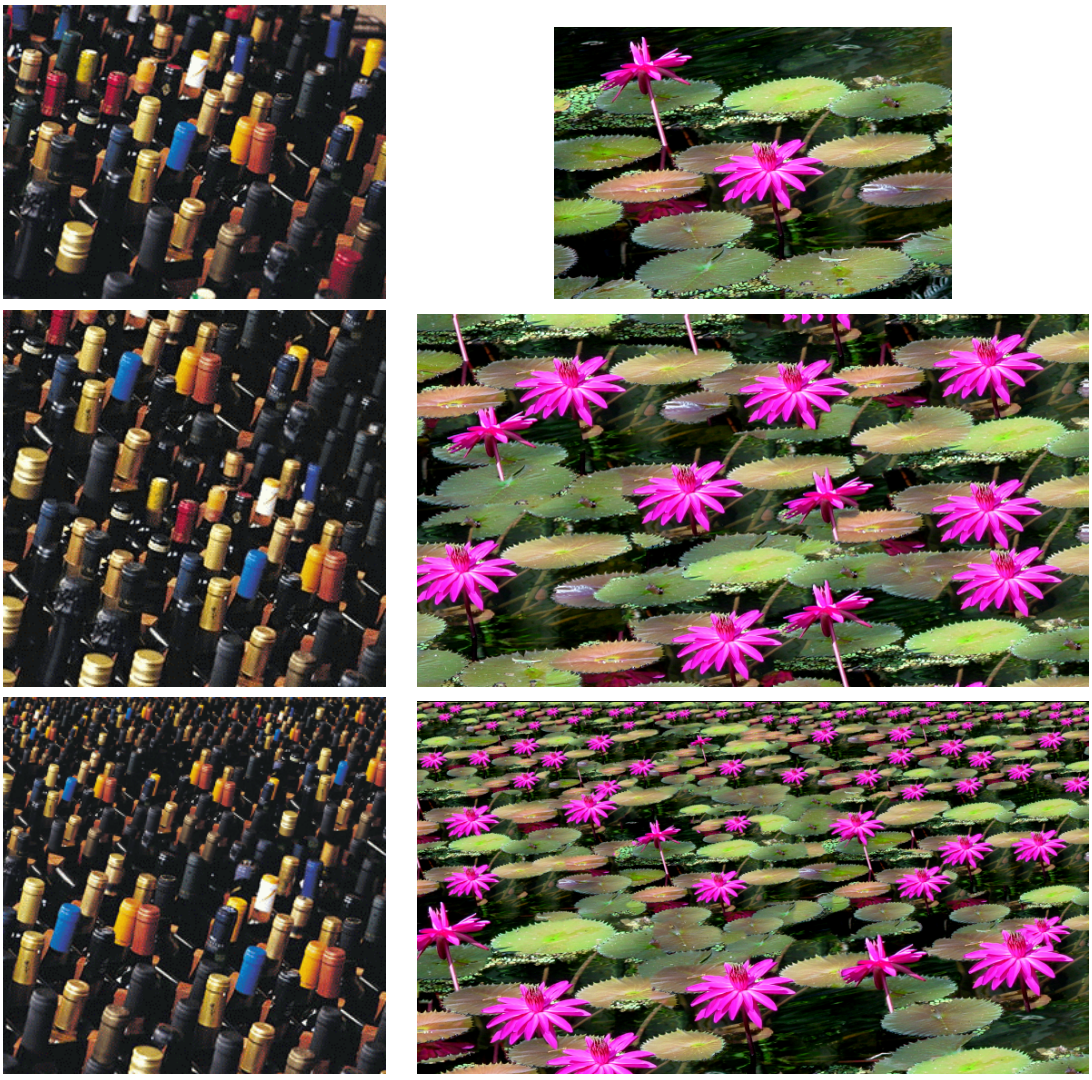


Figure 11: Images synthesized using multiple scales yield perspective effects. Shown are BOTTLES and LILIES©Brad Powell. We have, from top to bottom: input image, image synthesized using one scale, and image synthesized using multiple scales.

Figure 12: Example of interactive blending of two source images. Shown are inputs HUT and MOUNTAIN©Erskine Wood in the top row, and the result in the bottom row. In the bottom right image, the computed seams are also rendered on top of the results.

Figure 13: Another example of interactive blending of two source images. Shown are inputs RAFT and RIVER©Tim Seaver in the top row, and the result in the bottom row. In the bottom right image, the computed seams are also rendered on top of the results.

seam go through those regions. Note that our cost function as defined in (5) also favors the seam to go through edges. Our results indicate that both kinds of seams are present in the output images synthesized in this fashion. In the examples in Figure 12 and Figure 13, one can see that the seam goes through (a) the middle of the water which is the region of similarity between the source images, and (b) around the silhouettes of the people sitting in the raft which is a high gradient region.

The SIGGRAPH banner in Figure 14 was generated by combining flowers and leaves interactively: the user had to place a flower image over the leaves background and constrain some pixels of the output to come from within a flower. The graph cut algorithm was then used to compute the appropriate seam between the flower and the leaves automatically. Each letter of the word SIGGRAPH was synthesized individually and then these letters were

Figure 14: The SIGGRAPH banner at the top was generated by merging the source images shown below it. Image credits: (b)©East West Photo, (c)©Jens Grabenstein, (e)©Olga Zhaxybayeva.

combined, again using graph cuts, to form the final banner – the letter G was synthesized only once, and repeated. Approximate interaction time for each letter was in the range of 5-10 minutes.

It is worthwhile to mention related work on Intelligent Scissors by Mortensen and Barrett [42] in this context. They follow a two-step procedure of segmentation followed by composition to achieve similar effects. However, in our work, we don't segment the objects explicitly; instead we leave it to the cost function to choose between object boundaries and perceptually similar regions for the seam to go through. Also, the cost function used by them is different than ours. Perez et al. [44] on the other hand, blend pre-selected image regions using gradient-domain fusion.

3.5 Video Synthesis

One of the main strengths of the graph cut technique proposed here is that it allows for a straightforward extension to video synthesis. Consider a video sequence as a 3D collection of voxels, where one of the axes is time. Patches in the case of video are then the whole 3D space-time blocks of video, which can be placed anywhere in the 3D (space-time) volume. Hence, the same two steps from image texture synthesis, patch placement and seam finding, are also needed for video texture synthesis.

Similar to 2D texture, the patch selection method for video must be chosen based on the type of video. Some video sequences just show temporal stationarity whereas others show stationarity in space as well as time. For the ones showing only temporal stationarity, searching for patch translations in all three dimensions (space and time) is unnecessary. We can restrict our search just to patch offsets in time, *i.e.,* we just look for temporal translations of the patch. However, for videos that are spatially and temporally stationary, we do search in all three dimensions.

We now describe some of our video synthesis results. We start by showing some examples of temporally stationary textures in which we find spatio-temporal seams for video

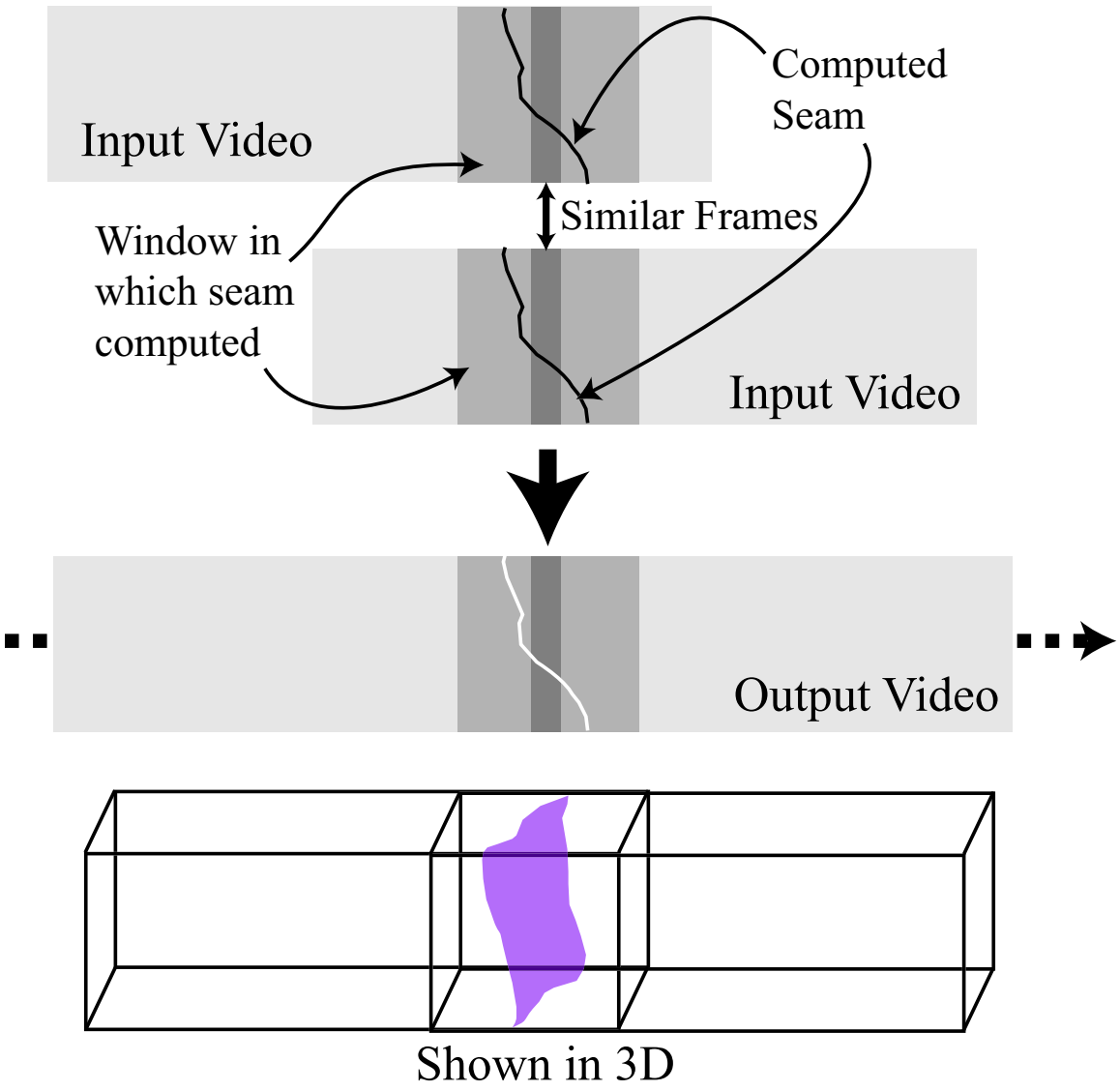


Figure 15: Illustration of seams for temporal texture synthesis. Seams shown in 2D and 3D for the case of video transitions. Note that the seam is a surface in case of video.

transitions. These results improve upon video textures [51] and compare favorably against dynamic textures [57]. Then we discuss spatio-temporally stationary type of video synthesis that improves upon [64, 2].

A. Finding Seams for Video Transitions: Video textures [51] turn existing video into an infinitely playing form by finding smooth *transitions* from one part of the video to another. These transitions are then used to infinitely loop the input video. This approach works only if a pair of similar-looking frames can be found. Many natural processes like fluids and small-scale motion are too chaotic for any frame to reoccur. To ease visual discontinuities due to frame mismatches, video textures used blending and morphing techniques. Unfortunately, a blend between transitions introduces an irritating blur. Morphing also does not work well for chaotic motions because it is hard to find corresponding features. Our seam optimization allows for a more sophisticated approach: the two parts of the video interfacing at a transition, represented by two 3D spatio-temporal texture patches, can be spliced together by computing the optimal seam between the two 3D patches. The seam in this case is actually a 2D surface that sits in 3D (Figure 15).

To find the best relative offset of the spatio-temporal texture patches, we first find a good transition by pair-wise image comparison as described in [51]. We then compute an optimal seam for a limited number of good transitions within a window around the transition. The result is equivalent to determining the time of the transition on a per-pixel basis rather than finding a single transition time for the whole image. The resulting seam can then be repeated to form a video loop as shown in Figure 15.

We have generated several (infinitely long) videos using this approach. For each sequence, we compute the optimal seam within a 60-frame spatio-temporal window centered around the best transition. Examples include WATERFALL A, GRASS, POND, FOUNTAIN, and BEACH. WATERFALL A and GRASS have been borrowed from Schödl et al. [51]. Their results on these sequences look intermittently blurred during the transition. Using our

technique, we are able to generate sequences without any perceivable artifacts around the transitions, which eliminates the need for any blurring. We have also applied (our implementation of) dynamic textures [57] to WATERFALL A, the result of which is much blurrier than our result. The BEACH example shows the limitations of our approach. Although the input sequence is rather long – 1421 frames – even the most similar frame pair does not allow a smooth transition. During the transition, a wave gradually disappears. Most disconcertingly, parts of the wave vanish from bottom to top, defying the usual dynamics of waves.

B. Random Temporal Offsets For very short sequences of video, looping causes very noticeable periodicity. In this case, we can synthesize a video by applying a series of input texture patches, which are randomly displaced in time. The seam is computed within the whole spatio-temporal volume of the input texture.

We have applied this approach to FIRE, SPARKLE, OCEAN, and SMOKE. The result for FIRE works relatively well and, thanks to random input patch displacements, is less repetitive than the comparable looped video. The SPARKLE result is also very nice, although electric sparks sometimes detach from the ball. In the case of OCEAN, the result is overall good, but the small amount of available input footage causes undesired repetitions. SMOKE is a failure of this method. There is no continuous part in this sequence that tiles well in time. Parts of the image appear almost static. The primary reason for this is the existence of a dominant direction of motion in the sequence, which is very hard to capture using temporal translations alone. Next, we discuss how to deal with such textures using spatio-temporal offsets.

C. Spatio-Temporally Stationary Videos: For videos that show spatio-temporal stationarity (like OCEAN and SMOKE), only considering translations in time does not produce good results. This is because, for such sequences, there usually is some dominant direction of motion for most of the spatial elements, that cannot be captured by just copying

pixels from different temporal locations; we need to move the pixels around in both space and time. We apply the sub-patch matching algorithm in 3D for spatio-temporal textures. Using such translations in space and time for spatio-temporal textures shows a remarkable improvement over using temporal translations alone.

The OCEAN sequence works very well with this approach, and the motion of the waves is quite natural. However, there are still slight problems with sea grass appearing and disappearing on the surface of the water. Even SMOKE shows a remarkable improvement. It is no longer static, as was the case with the previous method, showing the power of using spatio-temporal translations. RIVER, FLAME, and WATERFALL B also show very good results with this technique.

We have compared our results for FIRE, OCEAN, and SMOKE with those of Wei and Levoy [64] – we borrowed these sequences and their results from Wei and Levoy's web site. They are able to capture the local statistics of these temporal textures quite well but fail to reproduce their global structure. Our results show an improvement over them by being able to reproduce both the local and the global structure of the phenomena.

D. Temporal Constraints for Video One of the advantages of constraining new patch locations for video to temporal translations is that even though we find a spatio-temporal seam within a window of a few frames, the frames outside that window stay as is. This allows us to loop the video infinitely (see Figure 15). When we allow the translations to be in both space and time, this property is lost and it is non-trivial to make the video loop. It turns out, however, that we can use the graph cut algorithm to perform constrained synthesis (as in the case of interactive image merging) and therefore looping. We fix the first k and last k frames of the output sequence to be the same k frames of the input ($k = 10$ in our implementation). The pixels in these frames are now constrained to stay the same. This is ensured by adding links of infinite cost between these pixels and the patches they are constrained to copy from, during graph construction. The graph cut algorithm then

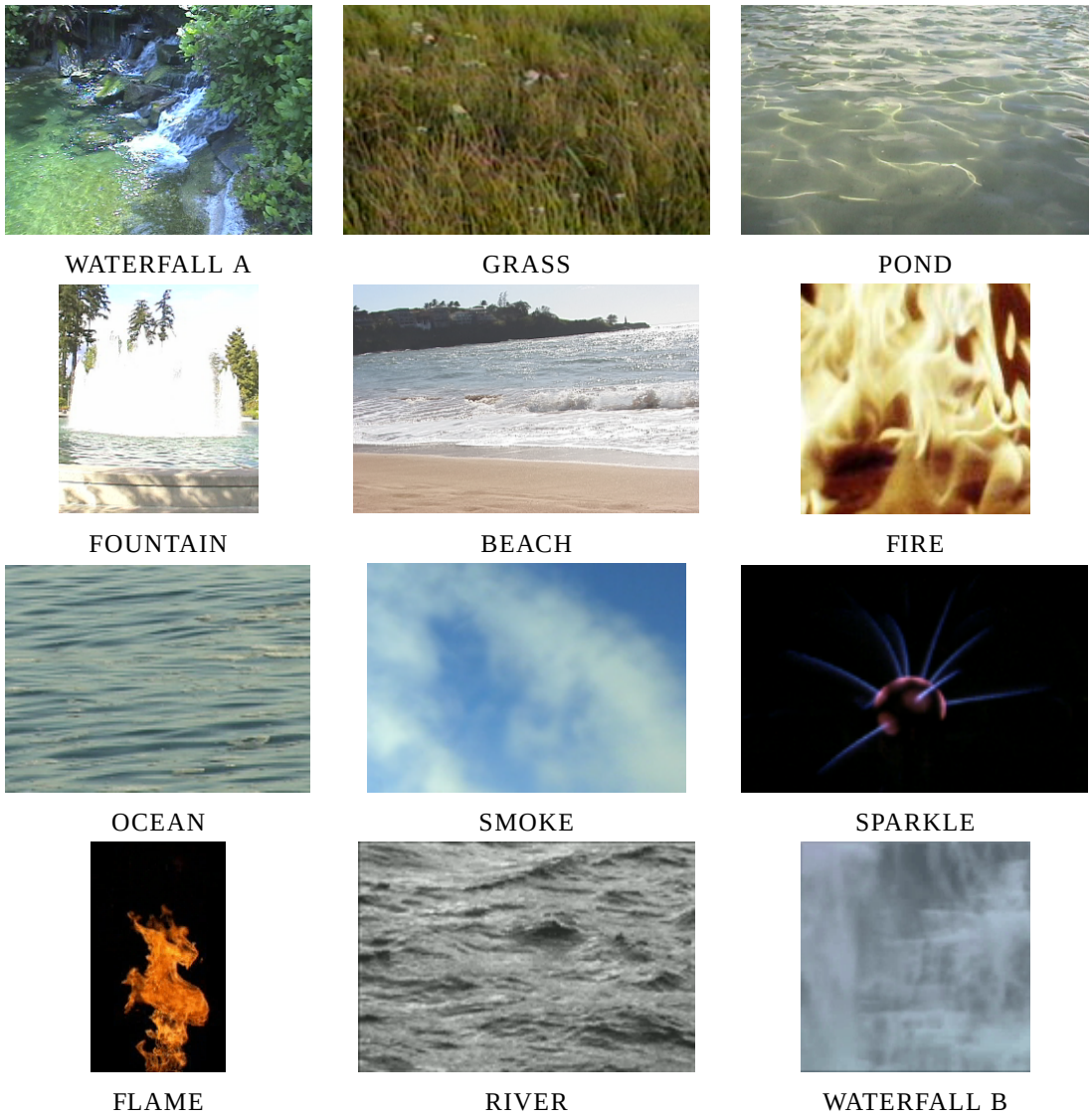


Figure 16: The various videos that we have synthesized using our approach.

computes the best possible seams given that these pixels don't change. Once the output has been generated, we remove the first k frames from it. This ensures a video loop since the k^{th} frame of the output is the same as its last frame before this removal operation.

Using this technique, we have been able to generate looped sequences for almost all of our examples. One such case is WATERFALL B, which was borrowed from [2]. We are able to generate an infinitely long sequence for this example, where as [2] can extend it to a finite length only. SMOKE is one example for which looping does not work very well.

E. Spatial Extensions for Video We can also increase the frame-size of the video sequence if we allow the patch translations to be in both space and time. We have been able to do so successfully for video sequences exhibiting spatio-temporal stationarity. For example, the spatial resolution of the RIVER sequence was increased from 170×116 to 210×160. By using temporal constraints, as explained in the previous paragraph, we were even able to loop this *enlarged* video sequence.

The running times for our video synthesis results ranged from 5 minutes to 1 hour depending on the size of video and the search method employed – searching for purely temporal offsets is faster than that for spatio-temporal ones. The use of FFT-based acceleration in our search algorithms was a huge factor in improving efficiency.

3.6 Summary

We have demonstrated a new algorithm for image and video synthesis. Our graph cut approach is ideal for computing seams of patch regions and determining placement of patches to generate perceptually smooth images and video. We have shown a variety of synthesis examples that include structured and random image and video textures. We also show extensions that allow for transformations of the input patches to permit variability in synthesis. We have also demonstrated an application that allows for merging of two different

source images interactively. In general, we believe that our technique significantly improves upon the state of the art in texture synthesis by providing the following benefits: (a) no restrictions on shape of the region where seam will be created, (b) consideration of old seam costs, (c) easy generalization to creation of seam surfaces for video, and (d) a simple method for adding pixel constraints.

The last point also leads to the remaining part of the thesis. While the approach presented in this chapter allows for constraints on pixel values, it is generally hard to provide a finer degree of control over the behavior of the synthesized texture, *e.g.,* it is conceivably impossible to synthesize a video using graph cuts in which the texture appears to follow a pre-specified flow field. The primary reason for this is that this approach is based on copying large patches to the output. Hence one patch placement step can change the appearance of a large number of pixels. This may not be desirable when one needs a finer level of control. Also, the graph cut approach lacks the notion of a metric that can measure the global quality of the synthesized texture – the seam cost is only a local measure and does not tell if the overall structures and elements present in the synthesized texture are consistent with the input. In the next few chapters, we generalize the notion of texture synthesis to example-based rendering of animations and present a general framework for designing such algorithms. We also develop a similarity metric for comparing input and output textures and extend it to incorporate metrics that measure consistency with control criteria. Techniques that optimize for these metrics are also presented.

CHAPTER IV

RENDERING ANIMATIONS USING TEXTURE EXEMPLARS

In this chapter, we propose a framework for using appearance information from example textures to render synthetic animations. The animation acts as a controller for the rendering process by guiding it to convey output characteristics like motion, shape, etc. This can also be viewed as a generalization of the texture synthesis work presented in Chapter 3. In the texture synthesis work, only the size and length of the output could be specified, while here we wish to provide a much finer degree of control through the use of animation characteristics.

We consider *phenomena that can be visually described as texture and dynamically described as fluid flow*. Water flowing down a stream, smoke rising from a chimney, and fire burning in a fireplace are all examples of such phenomena. We are interested in lending novel appearance to these phenomena by using different texture exemplars as sources. The current state-of-the-art in animating fluid phenomena is dominated by physical simulation techniques. These techniques use the physics of fluids to synthesize flow in space and time. Besides flow, other physical characteristics of the fluid like free-surface location and orientation, density, temperature, etc. can also be computed within the same simulation framework. Once this information is generated, the fluid is rendered using some desirable rendering technique like global illumination, sub-surface scattering, etc.

The choice of the rendering technique is more or less independent of the method used to synthesize the animation – it is usually based on simulation of an illumination model under known lighting conditions. The quality of synthesis depends on the photo-realism of

the rendering technique and accuracy of the parameters of the material being rendered. On the other hand, a sample of the fluid material imaged from the real world is a photo-realistic rendering of the phenomenon, by definition. Our goal is to exploit this fact by replacing the synthetic rendering step with an example-based rendering (EBR) step. This step maps appearance from the source texture (of fluid material) to target physical characteristics generated by the simulation.

One can situate example-based rendering in the context of the general problem of animation synthesis using example imagery (image or video). For example, starting with the video of a waterfall as example imagery, one might want to synthesize an animation that partially changes the flow of water in the video by adding obstacles in its path. In such a scenario, it is extremely desirable that the appearance of the edited (target) video match the appearance of the source video. Also, the target flow obtained after adding the obstacle should be physically plausible. In the following sections, we present a general framework for achieving this, *i.e.,* synthesizing animations by manipulating example imagery.

The focus of our research is the example-based rendering aspect of this framework. However, we explain the general animation synthesis framework to put our work in its broader context. Subsequently, we present a probabilistic formulation for constructing EBR algorithms within this framework.

4.1 Definitions

We first describe a few terms that will be frequently used in the rest of the chapter.

Source and Target: Source is used in reference to the example imagery that is employed for rendering the animation. The imagery could be an image or video of the phenomenon being animated. Target refers to the animation being rendered.

Animation Mechanism: The technique used for synthesizing the animation is referred to as animation mechanism. In our work, we have primarily used manual design/editing of

flow fields as the animation mechanism. However, it is not a restriction of our framework and might as well be any other technique, *e.g.*, physical simulation.

Rendering Variables: The appearance of an image depends upon various factors including lighting conditions, viewpoint location, intrinsic properties of the material being rendered (or imaged) like its bidirectional reflectance distribution function (BRDF) and texture, and its extrinsic properties like shape, density and temperature distribution, and flow, to name a few. We refer to these factors, whose complete knowledge is sufficient to definitively determine the appearance of an image or video, as rendering variables.

Characteristic: This term is used to refer to any physical characteristic that is either generated by the animation mechanism or extracted from the source exemplar, *e.g.*, flow, surface normal, density, temperature, etc. A characteristic does not have to be strictly physical; it may very well be an abstract characteristic that is supplied by the user. However, conceptually, characteristics are deemed as functions of rendering variables, and usually constitute a subset of the set of all rendering variables.

4.2 Animation Synthesis using Example Imagery

Given example (source) imagery of a particular phenomenon, synthesis of animation using that imagery requires (i) identifying characteristics of the phenomenon that define/represent the animation, (ii) understanding the underlying parameters that govern these characteristics, (iii) extracting characteristics and parameters from source imagery, (iv) modifying these parameters as desirable in order to generate new characteristics for the target animation, and (v) applying the example-based rendering step for producing target imagery that is similar to the source in appearance but consistent with the target characteristics at the same time. Note that the characteristics generated at the target may not all be available at the source. For example, if the source exemplar is an image, then it may not be possible to extract flow from it. However, the target animation may still be defined as a flow-field.

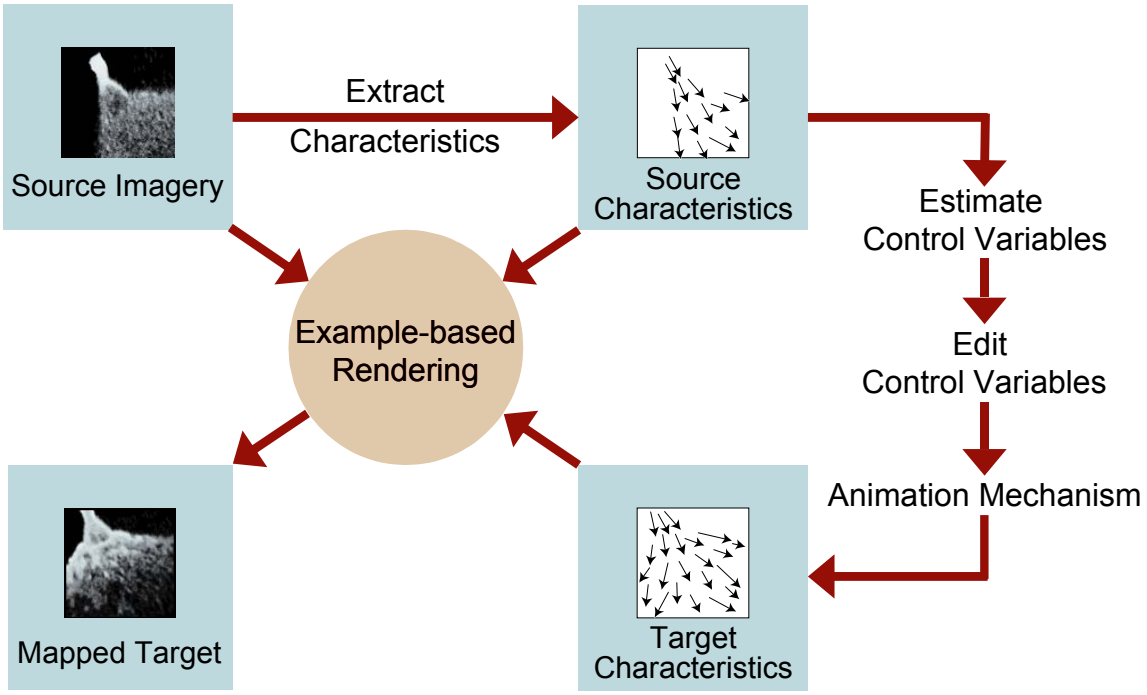


Figure 17: Animation Synthesis using Texture Exemplars. For explanation, see Section 4.2

In such cases, the consistency of target characteristics needs to be measured without the knowledge of source characteristics.

For fluid animation, examples of characteristics include flow, shape, density, temperature distribution, etc. These characteristics are governed by intrinsic parameters of the animated fluid and extrinsic structure of the scene. For example, viscosity is an intrinsic parameter of the fluid, while the external forces and obstacles form the extrinsic scene structure. For the waterfall scenario described earlier where we want to modify flow by adding obstacles, the steps for animation synthesis outlined above would require (i) extracting flow, viscosity and scene structure from the source video, (ii) modifying the scene structure to add an obstacle, followed by computing the corresponding target flow, and (iii) rendering the target using an example-based rendering algorithm. A framework for performing these steps algorithmically in the general case is shown schematically in Figure 17. A description of each step of the framework is as follows:

1. **Extraction of source characteristics:** A suite of algorithms for extracting characteristics should first be applied to the source exemplar to compute various characteristics; we call these *extracted source characteristics*. For example, one might use an optic flow computation algorithm to determine source flow, or a shape from shading algorithm to determine shape.

2. **Parameter & structure estimation:** The goal of parameter & structure estimation (Figure 18) is to compute control variables (parameters and structure) governing the example phenomenon using information from source characteristics. One way of determining whether an instance of control variables mimics the source is to synthesize source characteristics using the control variables and then compare them with the extracted source characteristics. For this, one requires a generative model for synthesizing characteristics from control variables and also needs the characteristic extraction procedure to be trustworthy. One can use this generate and test methodology to do a search for the control variables. However, the space of possible parameters and structures is usually too large to search exhaustively. Hence, a more efficient approach for estimating the control variables needs to be explored on a case-by-case basis.

3. **Generating target parameters and characteristics:** Once the control variables for the source have been estimated, they need to be edited to generate the target control variables. For example, in the case of fluid animation, one might change the viscosity of the fluid, modify external forces, or add/remove obstacles. The framework is relatively agnostic about the nature of this editing procedure. These parameters are then be passed to the animation mechanism in order to synthesize the target animation and corresponding characteristics. If the parameters used are physical in nature, *e.g.,* viscosity, then simulation would be an appropriate animation mechanism for generating target characteristics. However, it could also be an interactive procedure,

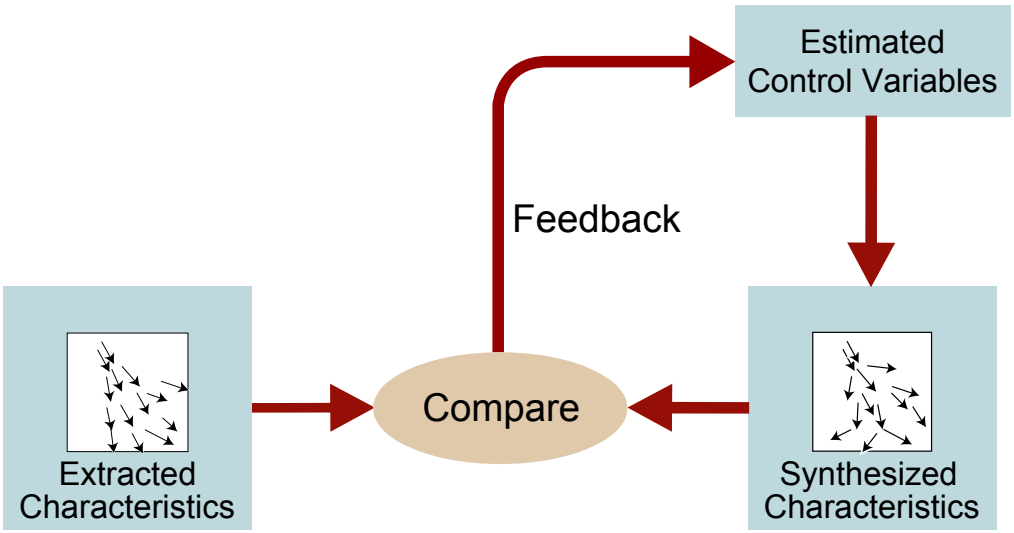


Figure 18: Parameter & Structure Estimation. Shown is a generate and test methodology for searching control variables. The synthesized and extracted characteristics are being compared for consistency inside a feedback loop.

where the target characteristics are manually (interactively) designed or generated by editing the source characteristics.

4. **Example-based rendering of the target:** The role of the EBR algorithm (Figure 19) within this framework is to generate appearance for the target animation using information from source imagery. This step of the framework is the primary focus of our research. The EBR algorithm can be considered as an independent module in our framework; it is oblivious to the exact nature of the parameter & structure estimation algorithm or the editing procedure used to generate target characteristics. Note that the target animation may be synthesized in 3D space, but the eventual rendering is done from the vantage of a particular viewpoint. We assume that the animation mechanism is able to synthesize target characteristics as projected onto this particular viewpoint. For example, if the synthesized characteristic is flow, then the animation mechanism would *render* the 2D target flow as seen by the viewpoint of interest in the target image plane. The EBR algorithm would then find a mapping from the rendered

characteristics to appearance values in order to generate a rendering of the target – we call it the *mapped target*. In the process, it may make use of extracted source characteristics to generate the mapping. An important observation about the mapping from target characteristics to source appearance is that it may be multi-valued, *i.e.*, the same target characteristic value may be mapped to different source appearance values. Intuitively, this is a consequence of two potentially conflicting goals of any EBR algorithm: it needs to be as close as possible to the source in appearance space, and as close as possible to the target in characteristic space. The next section describes these two objectives mathematically using a probabilistic formulation.

The framework presented here is general in terms of the characteristics it can handle. However, the choice of the characteristic may determine the exact nature of the EBR algorithm or parameter estimation procedure used. Note that the source characteristics used by parameter & structure estimation may be different from those supplied to the EBR algorithm. For example, the estimation algorithm may use surface normals to determine appropriate parameters, while the rendering algorithm may be provided a flow-field for appearance mapping.

The subsequent sections and chapters focus on the example-based rendering step of the animation synthesis framework. Some aspects of target characteristic generation using interactive editing are also discussed. Even though extraction of source characteristics as well as automatic parameter estimation and editing are important components of the framework, we envision that they require considerable research effort of their own and therefore leave them for future work.

4.3 Probabilistic Formulation

The EBR algorithm needs to compute appearance values for the target, given source appearance, target characteristics and (optionally) source characteristics. Specific algorithms will

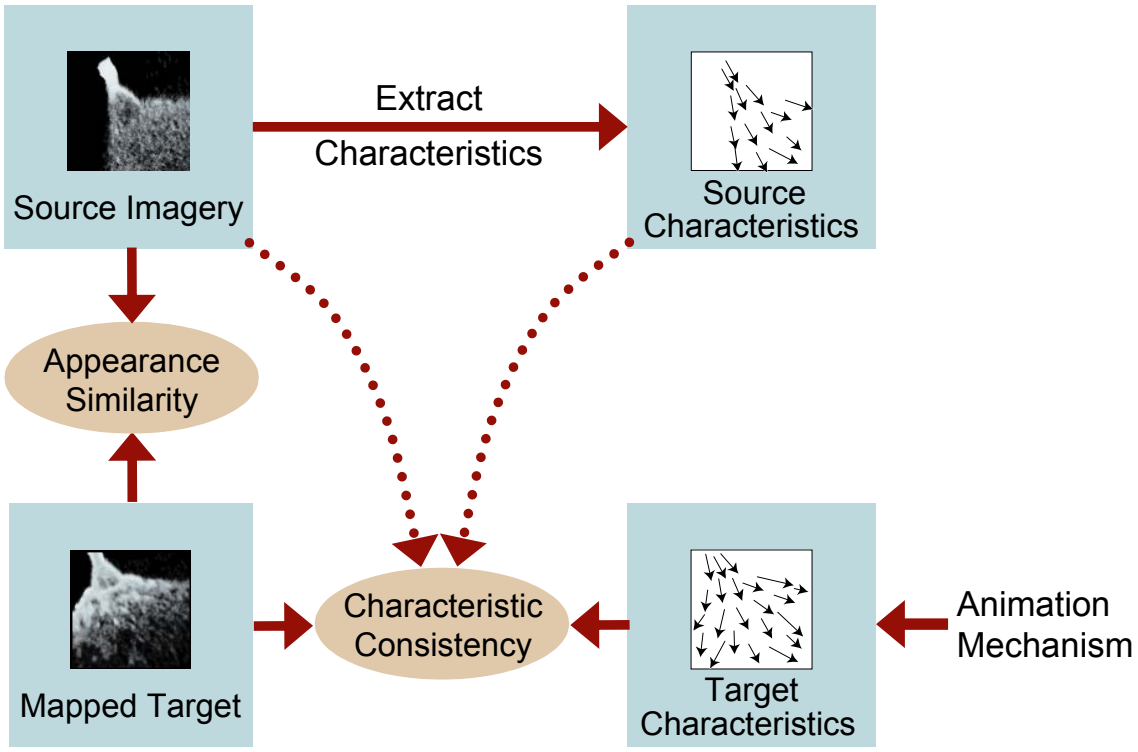


Figure 19: Example-based Rendering. The source imagery, source characteristics and target characteristics form the input to the EBR algorithm. The output of this algorithm is the mapped target. The target is compared with source imagery to measure appearance similarity, and with all inputs to measure consistency of characteristics.

depend on the nature of given characteristics and exemplars. However, the general principles governing their design may be the same. Here, we develop a probabilistic formulation to aid the design of general EBR algorithms.

Notation: We denote the source and target by $\mathcal{S}$ and $\mathcal{T}$, respectively. The rendering variables are collectively denoted by $\mathbf{V}$, and are defined on a per-pixel basis: $\mathbf{V}(p)$ contains the values of the rendering variables as observed at pixel p. This notation naturally extends to a set of pixels $\mathcal{P}$:

$$\mathbf{V}(\mathcal{P}) = \bigcup_{p \in \mathcal{P}} \mathbf{V}(p).$$

Characteristics are denoted by $\mathbf{C}$. They constitute the subsets of $\mathbf{V}$ (or functions of it) that are either extracted from the source or rendered at the target. The appearance (*i.e.,* pixel intensities) of an image or video is denoted by E. As in the case of $\mathbf{V}$, we denote the characteristics or appearance at pixel p (set of pixels $\mathcal{P}$) by $\mathbf{C}(p)$ or $E(p)$ ($\mathbf{C}(\mathcal{P})$ or $E(\mathcal{P})$), as appropriate. When used without a pixel or set index, the corresponding variable represents the entire *field* of values taken by that variable in the source or target domain. Additionally, when necessary, we specialize these variables using subscripts $\mathcal{S}$ and $\mathcal{T}$ to distinguish between the source and the target. Hence, $\mathbf{V}_\mathcal{S}$, $\mathbf{C}_\mathcal{S}$, and $E_\mathcal{S}$ denote the source variables, while $\mathbf{V}_\mathcal{T}$, $\mathbf{C}_\mathcal{T}$, and $E_\mathcal{T}$ denote the target variables.

4.3.1 Assumptions

Since we want to use the source imagery to render the target, it is desirable that the source and target rendering variables be as close as possible. In the limit when all variables are the same, the appearance of the target is identical to the source. On the other hand, if all variables are free to be arbitrarily different in the source and the target, it may be impossible to infer the appearance of the target from the source – for example, the source and target may consist of different materials in which case the source appearance is unusable for the target. Hence, in order to make the problem tractable, we make a few simplifying

assumptions on the nature of rendering variables:

1. **Material assumption:** The material being rendered in the target is same as the one imaged in the source. This constrains the source and the target to have similar intrinsic properties like BRDF and texture.

2. **Lighting and Viewing assumption:** The lighting and viewing conditions are same for both the source and the target. The consequence of this assumption is that the source and target can be considered part of the same scene. Effectively, we are assuming that, given sufficient knowledge of the rest of the rendering variables, the effect of lighting and viewing conditions may be ignored.

3. **Locality & Homogeneity assumption:** The values of rendering variables at a particular pixel completely determine the intensity at that pixel. In other words, the appearance at a pixel is a function of only the rendering variables at that pixel. Also, this function is homogenous w.r.t. position, *i.e.*, it is independent of the location of the pixel. If we denote this function by f, then for all pixels $p \in \mathcal{S}, \mathcal{T}$

$$E(p) = f(\mathbf{V}(p))$$

which can be probabilistically written as

$$P(E(p)|\mathbf{V}(p)) = \delta(E(p) - f(\mathbf{V}(p))).$$

Note that the function f is the same for both the source and the target. This is a consequence of our previous assumptions that the material, lighting, and viewpoint are common for both the source and the target. We explicitly express this commonality of f to $\mathcal{S}$ and $\mathcal{T}$ as

$$P(E_{\mathcal{T}}(p)|\mathbf{V}_{\mathcal{T}}(p) = v) = P(E_{\mathcal{S}}(p)|\mathbf{V}_{\mathcal{S}}(p) = v). \quad (7)$$

The actual location of the pixel p in this equation is irrelevant because of the homogeneity assumption. These assumptions are necessary for making the synthesis of

target appearance computationally tractable; we use them when defining similarity measures between source and target appearance, and consistency measures between target characteristics and target appearance. Assuming locality and homogeneity also imposes certain limitations: we lose the ability to directly reason about or synthesize global illumination effects like inter-reflections, shadows, and refraction through air. We make the following assumption to partially compensate for these limitations.

4. **Distribution assumption:** The rendering variables in a pixel neighborhood follow a joint probability distribution, which is presumed responsible for any observed structure in the appearance of neighborhoods. We don't assume knowledge of this distribution; only that it is the same for both the source and the target rendering variables. Mathematically,

$$P(\mathbf{V}_{\mathcal{T}}(\mathcal{N}_p) = \mathbf{v}) = P(\mathbf{V}_{\mathcal{S}}(\mathcal{N}_p) = \mathbf{v}) \tag{8}$$

where, $\mathcal{N}_p$ is a neighborhood of pixels centered at p. This assumption allows us to implicitly model short-range interactions between neighboring pixels and consequently, between corresponding points on the material.

5. **Markov Random Field (MRF) assumption:** The appearance of both the source and the target behaves like a spatio-temporal texture, which is modeled as a Markov Random Field. In a texture modeled as an MRF, the intensity at a particular pixel depends only on the intensity of a small set of neighboring pixels. Mathematically,

$$P(E(p)|E(\bar{p})) = P(E(p)|E(\mathcal{N}_p \backslash p)) \tag{9}$$

where $\bar{p}$ denotes the set of all the pixels in the domain other than p, and $\mathcal{N}_p \backslash p$ denotes the set of pixels in the neighborhood $\mathcal{N}_p$ other than the pixel p itself. This assumption, along with the locality, homogeneity, and distribution assumptions, is useful for computing similarity between source and target appearance in a tractable fashion.

4.3.2 Appearance Estimation

From a probabilistic point of view, we want to synthesize the most probable target appearance, given the target characteristics, the source appearance and the source characteristics. The maximum *a posteriori* (MAP) estimate of $E_{\mathcal{T}}$ is

$$\widehat{E}_{\mathcal{T}} = \arg\max_{E_{\mathcal{T}}} P(E_{\mathcal{T}}|\mathbf{C}_{\mathcal{T}}, E_{\mathcal{S}}, \mathbf{C}_{\mathcal{S}}).$$

This posterior implicitly encodes two objectives: in terms of characteristics, we want $E_{\mathcal{T}}$ to be consistent with $\mathbf{C}_{\mathcal{T}}$, while in terms of appearance, we want $E_{\mathcal{T}}$ to be close to $E_{\mathcal{S}}$. We can express these goals explicitly, by applying the Bayes rule:

$$P(E_{\mathcal{T}}|\mathbf{C}_{\mathcal{T}}, E_{\mathcal{S}}, \mathbf{C}_{\mathcal{S}}) = \frac{P(\mathbf{C}_{\mathcal{T}}|E_{\mathcal{T}}, E_{\mathcal{S}}, \mathbf{C}_{\mathcal{S}}).P(E_{\mathcal{T}}|E_{\mathcal{S}}, \mathbf{C}_{\mathcal{S}})}{P(\mathbf{C}_{\mathcal{T}}|E_{\mathcal{S}}, \mathbf{C}_{\mathcal{S}})}.$$

The normalization term, $P(\mathbf{C}_{\mathcal{T}}|E_{\mathcal{S}}, \mathbf{C}_{\mathcal{S}})$, is constant for all $E_{\mathcal{T}}$. Hence, it can be ignored for MAP estimation:

$$P(E_{\mathcal{T}}|\mathbf{C}_{\mathcal{T}}, E_{\mathcal{S}}, \mathbf{C}_{\mathcal{S}}) \propto P(\mathbf{C}_{\mathcal{T}}|E_{\mathcal{T}}, E_{\mathcal{S}}, \mathbf{C}_{\mathcal{S}}).P(E_{\mathcal{T}}|E_{\mathcal{S}}, \mathbf{C}_{\mathcal{S}}).$$

Of the two terms on the RHS, the first term, $P(\mathbf{C}_{\mathcal{T}}|E_{\mathcal{T}}, E_{\mathcal{S}}, \mathbf{C}_{\mathcal{S}})$, is the likelihood, and measures the consistency of the synthesized target appearance with the target characteristics. The second term, $P(E_{\mathcal{T}}|E_{\mathcal{S}}, \mathbf{C}_{\mathcal{S}})$ represents the prior, and measures the closeness between the source and the target in appearance space (see Figure 20). These terms also contain $\mathbf{C}_{\mathcal{S}}$ as a conditioning variable. In the case of likelihood, $\mathbf{C}_{\mathcal{S}}$ may be useful for learning a relationship between appearance and characteristics, using $E_{\mathcal{S}}$ and $\mathbf{C}_{\mathcal{S}}$ as training data. However, for the prior, we assume that $\mathbf{C}_{\mathcal{S}}$ does not provide any extra information and drop it from the expression. This is reasonable under our interpretation of the prior that it measures proximity only in appearance space. Hence, we obtain the following expression for the MAP estimate:

$$\widehat{E}_{\mathcal{T}} = \arg\max_{E_{\mathcal{T}}} P(\mathbf{C}_{\mathcal{T}}|E_{\mathcal{T}}, E_{\mathcal{S}}, \mathbf{C}_{\mathcal{S}}).P(E_{\mathcal{T}}|E_{\mathcal{S}}). \tag{10}$$

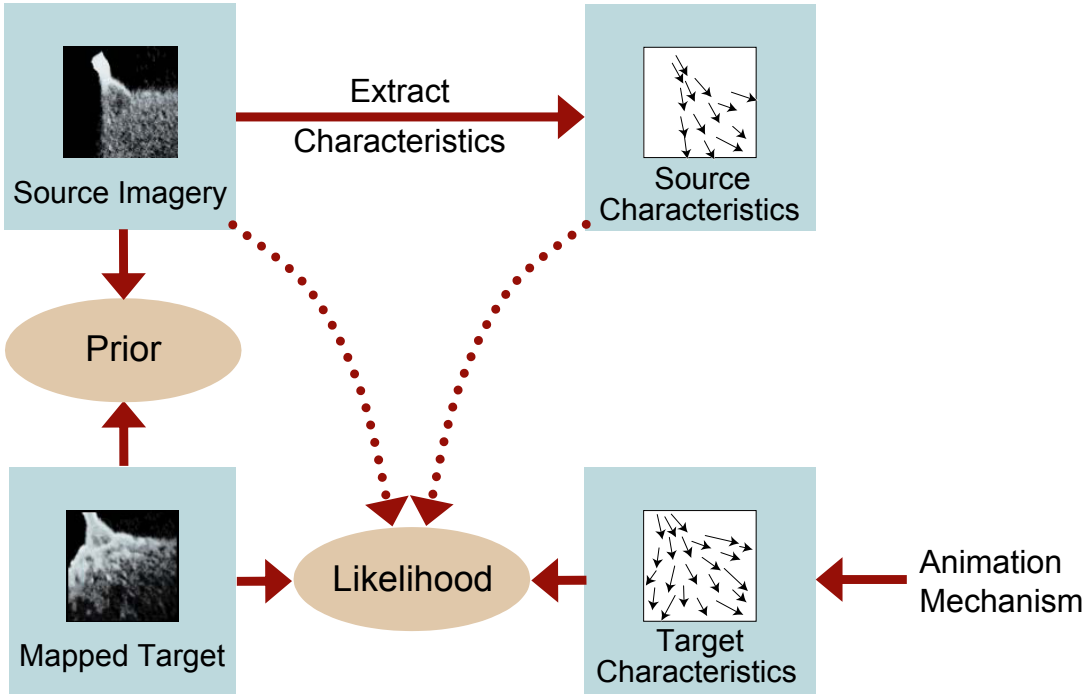


Figure 20: The appearance similarity and characteristic consistency operations of the EBR algorithm are formulated as prior and likelihood, respectively (also see Figure 19 for comparison).

Prior: The prior predicts the target appearance without any knowledge of the rendering variables at the target, $\mathbf{V}_{\mathcal{T}}$. This capability is significant because of two reasons. Firstly, we have only incomplete knowledge of $\mathbf{V}_{\mathcal{T}}$ – in the form of characteristics, $\mathbf{C}_{\mathcal{T}}$ – which is not sufficient to deterministically obtain $E_{\mathcal{T}}$. Secondly, even if we knew $\mathbf{V}_{\mathcal{T}}$ completely, the function that maps $\mathbf{V}_{\mathcal{T}}$ to $E_{\mathcal{T}}$ is unknown. In order to establish a prior under such under-determined conditions, we will invoke the assumptions made earlier in this section. We start by applying the MRF assumption. Consider the probability of observing the intensity $E_{\mathcal{T}}(p)$ at a target pixel p, given the intensities at all other pixels in the target. Then, according to the MRF assumption (9) applied to $\mathcal{T}$,

$$P(E_{\mathcal{T}}(p)|E_{\mathcal{T}}(\bar{p})) = P(E_{\mathcal{T}}(p)|E_{\mathcal{T}}(\mathcal{N}_p \backslash p)).$$

We can marginalize the RHS over all possible target rendering variables in $\mathcal{N}_p$, to get

$$\begin{aligned} P(E_{\mathcal{T}}(p)|E_{\mathcal{T}}(\mathcal{N}_p\backslash p)) &= \sum_{\mathbf{V}_{\mathcal{T}}(\mathcal{N}_p)} P(E_{\mathcal{T}}(p),\mathbf{V}_{\mathcal{T}}(\mathcal{N}_p)|E_{\mathcal{T}}(\mathcal{N}_p\backslash p)) \\ &= \sum_{\mathbf{V}_{\mathcal{T}}(\mathcal{N}_p)} P(E_{\mathcal{T}}(p)|\mathbf{V}_{\mathcal{T}}(\mathcal{N}_p),E_{\mathcal{T}}(\mathcal{N}_p\backslash p)).P(\mathbf{V}_{\mathcal{T}}(\mathcal{N}_p)|E_{\mathcal{T}}(\mathcal{N}_p\backslash p)). \end{aligned} \tag{11}$$

The first term inside the summation in (11), $P(E_{\mathcal{T}}(p)|\mathbf{V}_{\mathcal{T}}(\mathcal{N}_p),E_{\mathcal{T}}(\mathcal{N}_p\backslash p))$, is the probability of observing the target intensity $E_{\mathcal{T}}(p)$ at pixel p, given the rendering variables $\mathbf{V}_{\mathcal{T}}(\mathcal{N}_p)$ in the neighborhood $\mathcal{N}_p$ around that pixel and also the rest of the intensities $E_{\mathcal{T}}(\mathcal{N}_p\backslash p)$ in that neighborhood. According to our locality assumption, $E_{\mathcal{T}}(p)$ is independent of all other variables if $\mathbf{V}_{\mathcal{T}}(p)$ is known. Therefore,

$$P(E_{\mathcal{T}}(p)|\mathbf{V}_{\mathcal{T}}(\mathcal{N}_p),E_{\mathcal{T}}(\mathcal{N}_p\backslash p)) = P(E_{\mathcal{T}}(p)|\mathbf{V}_{\mathcal{T}}(p)). \tag{12}$$

The second term inside the summation in (11), $P(\mathbf{V}_{\mathcal{T}}(\mathcal{N}_p)|E_{\mathcal{T}}(\mathcal{N}_p\backslash p))$, can be rewritten after applying the Bayes rule, as

$$\begin{aligned} P(\mathbf{V}_{\mathcal{T}}(\mathcal{N}_p)|E_{\mathcal{T}}(\mathcal{N}_p\backslash p)) &= \frac{P(E_{\mathcal{T}}(\mathcal{N}_p\backslash p)|\mathbf{V}_{\mathcal{T}}(\mathcal{N}_p)).P(\mathbf{V}_{\mathcal{T}}(\mathcal{N}_p))}{P(E_{\mathcal{T}}(\mathcal{N}_p\backslash p))} \\ &= \frac{P(E_{\mathcal{T}}(\mathcal{N}_p\backslash p)|\mathbf{V}_{\mathcal{T}}(\mathcal{N}_p)).P(\mathbf{V}_{\mathcal{T}}(\mathcal{N}_p))}{\sum_{\mathbf{V}_{\mathcal{T}}(\mathcal{N}_p)} P(E_{\mathcal{T}}(\mathcal{N}_p\backslash p)|\mathbf{V}_{\mathcal{T}}(\mathcal{N}_p)).P(\mathbf{V}_{\mathcal{T}}(\mathcal{N}_p))}. \end{aligned} \tag{13}$$

The term $P(E_{\mathcal{T}}(\mathcal{N}_p\backslash p)|\mathbf{V}_{\mathcal{T}}(\mathcal{N}_p))$ in (13) measures the probability of pixel intensities in the neighborhood $\mathcal{N}_p$ (excluding p itself), given the rendering variables in that neighborhood. The locality assumption can also be applied to this term, which reduces it to

$$P(E_{\mathcal{T}}(\mathcal{N}_p\backslash p)|\mathbf{V}_{\mathcal{T}}(\mathcal{N}_p)) = \prod_{q\in\mathcal{N}_p\backslash p} P(E_{\mathcal{T}}(q)|\mathbf{V}_{\mathcal{T}}(q)). \tag{14}$$

We can substitute the simplifications from (12), (13), and (14) into (11) to obtain the following equation:

$$P(E_{\mathcal{T}}(p)|E_{\mathcal{T}}(\mathcal{N}_p\backslash p)) = \frac{\sum_{\mathbf{V}_{\mathcal{T}}(\mathcal{N}_p)} \prod_{q\in\mathcal{N}_p} P(E_{\mathcal{T}}(q)|\mathbf{V}_{\mathcal{T}}(q)).P(\mathbf{V}_{\mathcal{T}}(\mathcal{N}_p))}{\sum_{\mathbf{V}_{\mathcal{T}}(\mathcal{N}_p)} \prod_{q\in\mathcal{N}_p\backslash p} P(E_{\mathcal{T}}(q)|\mathbf{V}_{\mathcal{T}}(q)).P(\mathbf{V}_{\mathcal{T}}(\mathcal{N}_p))}. \tag{15}$$

The only difference between the numerator and denominator in (15) is that the product in the numerator includes all pixels in the neighborhood $\mathcal{N}_p$ while the denominator excludes the term corresponding to pixel p itself. We further recall that the function f that governs the relationship between $E(p)$ and $\mathbf{V}(p)$ is the same for both $\mathcal{S}$ and $\mathcal{T}$ (see (7)). Also, according to our distribution assumption, $\mathbf{V}_{\mathcal{T}}(\mathcal{N}_p)$ follows the same distribution as $\mathbf{V}_{\mathcal{S}}(\mathcal{N}_p)$ (see (8)). Incorporating this information into (15), we get

$$P(E_{\mathcal{T}}(p)|E_{\mathcal{T}}(\mathcal{N}_p\backslash p)) = \frac{\Sigma_{\mathbf{V}_{\mathcal{S}}(\mathcal{N}_p)} \prod_{q\in\mathcal{N}_p} P(E_{\mathcal{S}}(q) = E_{\mathcal{T}}(q)|\mathbf{V}_{\mathcal{S}}(q)).P(\mathbf{V}_{\mathcal{S}}(\mathcal{N}_p))}{\Sigma_{\mathbf{V}_{\mathcal{S}}(\mathcal{N}_p)} \prod_{q\in\mathcal{N}_p\backslash p} P(E_{\mathcal{S}}(q) = E_{\mathcal{T}}(q)|\mathbf{V}_{\mathcal{S}}(q)).P(\mathbf{V}_{\mathcal{S}}(\mathcal{N}_p))}.$$

We can reverse the steps between (11) and (15) and apply them to the RHS to get the following result:

$$P(E_{\mathcal{T}}(p)|E_{\mathcal{T}}(\mathcal{N}_p\backslash p)) \;=\; P(E_{\mathcal{S}}(p) = E_{\mathcal{T}}(p)|E_{\mathcal{S}}(\mathcal{N}_p\backslash p) = E_{\mathcal{T}}(\mathcal{N}_p\backslash p)),$$

or equivalently

$$P(E_{\mathcal{T}}(p) = e|E_{\mathcal{T}}(\mathcal{N}_p\backslash p) = \mathbf{e}) \;=\; P(E_{\mathcal{S}}(p) = e|E_{\mathcal{S}}(\mathcal{N}_p\backslash p) = \mathbf{e}). \tag{16}$$

This is the key equation for our prior. It says that the conditional probability of observing a particular intensity at a pixel, given its neighbors, is the same for both the source and the target. Additionally, this probability is independent of the location of the pixel. Since the source appearance is known, we can use it to predict the appearance of the target using (16). In fact, this prior reflects the same assumptions as are made by MRF based texture synthesis algorithms. If we remove the likelihood from the objective function – this would happen when no target characteristics are available – the problem reduces to that of texture synthesis. Our formulation generalizes the notion of texture synthesis to incorporate the handling of desirable target characteristics. In Chapter 5, we present a novel technique for texture synthesis based on optimization of the prior term alone.

Likelihood: As mentioned above, likelihood measures the consistency of synthesized target appearance with desirable target characteristics. Semantically, this notion of likelihood

is slightly different from its conventionally accepted interpretation. Usually, likelihood refers to the probability of making an observation, given the hidden variables governing the generation of that observation. In other words, there is an underlying generative model that predicts the observation from the given hidden variables. Likelihood then measures the similarity between the predicted and measured observations. Under this convention, $\mathbf{C}_{\mathcal{T}}$ is the observation, while $E_{\mathcal{T}}$ is the hidden variable. However, in reality, $\mathbf{C}_{\mathcal{T}}$ is not an experimental observation. It is a desired value explicitly set by the animation mechanism. $E_{\mathcal{T}}$, on the other hand, *induces* a characteristic field over the target. The induced characteristic field is the characteristic field perceived from the synthesized target appearance $E_{\mathcal{T}}$ – what one may obtain by running the characteristic extraction algorithm on $E_{\mathcal{T}}$ instead of $E_{\mathcal{S}}$. The goal of the EBR algorithm is to infer an $E_{\mathcal{T}}$ for which the induced characteristics at the target are as close as possible to $\mathbf{C}_{\mathcal{T}}$. Thus, likelihood is interpreted as a measure of similarity between the induced and the desired characteristics. The actual form of this likelihood and the corresponding EBR algorithm will depend on the characteristics in question.

The computation of likelihood may not necessarily use $E_{\mathcal{S}}$ or $\mathbf{C}_{\mathcal{S}}$, in which case the expression would simplify, as appropriate, to $P(\mathbf{C}_{\mathcal{T}}|E_{\mathcal{T}},\mathbf{C}_{\mathcal{S}})$, $P(\mathbf{C}_{\mathcal{T}}|E_{\mathcal{T}},E_{\mathcal{S}})$, or $P(\mathbf{C}_{\mathcal{T}}|E_{\mathcal{T}})$. One reason for this could be that the characteristic of interest may be difficult or even impossible to estimate at the source. The primary benefit of having the knowledge of source characteristics is that one can use it to establish a relationship between characteristics and the appearance. This relationship can then be used to map target characteristics to target appearance. In the scenario where source characteristics are not available, one needs to use prior domain knowledge about the specific characteristic to measure the consistency between appearance and characteristics.

Later, in Chapter 6 and Chapter 7, we discuss the design of likelihood functions for the special case where flow is the target characteristic, and image and video textures are source exemplars. Note that flow is not defined at the source if the exemplar used is an image texture. Our EBR algorithm takes this into account by defining likelihood using only target

flow and target appearance. In the case of video exemplars, source flow is meaningful. However, it is usually difficult to compute it accurately for textural videos, because the dynamics of appearance change in the video is a complex combination of texture evolution and flow. Hence, even in the case of video, we avoid the explicit computation of source flow. Instead, we have developed a technique that implicitly searches for source regions with flow similar to that desired at the target.

4.4 Summary

In this chapter, example-based rendering has been presented in the general context of synthesizing animations using texture imagery. We treat example-based rendering as a mechanism for rendering target animations with pre-specified characteristics like flow, shape, etc. Under this interpretation, an EBR algorithm should not only maintain similarity between target and source appearance, but also make sure that the synthesized target is consistent with the desirable characteristics. This generalizes the notion of texture synthesis as presented in the previous chapter. We have presented a probabilistic formulation for the design of EBR algorithms in general. This formulation casts synthesis as maximum *a posteriori* (MAP) appearance estimation, where appearance similarity serves as the prior and characteristic consistency serves as the likelihood. In the next few chapters, we present specific techniques designed using this formulation. The next chapter deals with synthesis using the prior term only, which leads to a novel optimization-based algorithm for texture synthesis. In chapters after that, we incorporate likelihood into our synthesis algorithm. Specifically, our experimentation consists of using flow as a characteristic and image/video exemplars as source appearance.

CHAPTER V

TEXTURE OPTIMIZATION FOR UNCONSTRAINED SYNTHESIS

In the last chapter, we developed general principles for example-based rendering. We now present a specific technique that makes use of those principles. Recall that we formulated appearance similarity and characteristic consistency as prior and likelihood in a probabilistic appearance estimation framework. In this chapter, we present a technique for texture synthesis based on specializing the probabilistic formulation to only consider the prior term, *i.e.*, we ignore characteristics and only optimize for appearance similarity – shown schematically in Figure 21. The next chapter extends this technique to also consider characteristics (with a specific implementation for flow). A related publication is [34].

The texture synthesis technique presented here is based on optimization over an appearance similarity metric. This similarity metric is motivated by the Markov Random Field (MRF)-based similarity criterion used in most local pixel-based synthesis techniques. Our contribution is to merge these locally defined similarity measures into a global metric that can be used to jointly optimize the entire texture. This global metric allows modeling of interactions between large neighborhoods; nevertheless, it can be optimized using a simple iterative algorithm with reasonable computational cost.

We make use of the fact that maximum *a posteriori* estimation over a probability function can be transformed into an equivalent optimization over an energy function. If $P(\mathbf{x})$ is the probability function, then $U(\mathbf{x}) = -\log P(\mathbf{x})$ is the equivalent energy function. We define an energy function for measuring the quality of the synthesized texture with respect to a given input texture by comparing local neighborhoods in the two textures. This energy

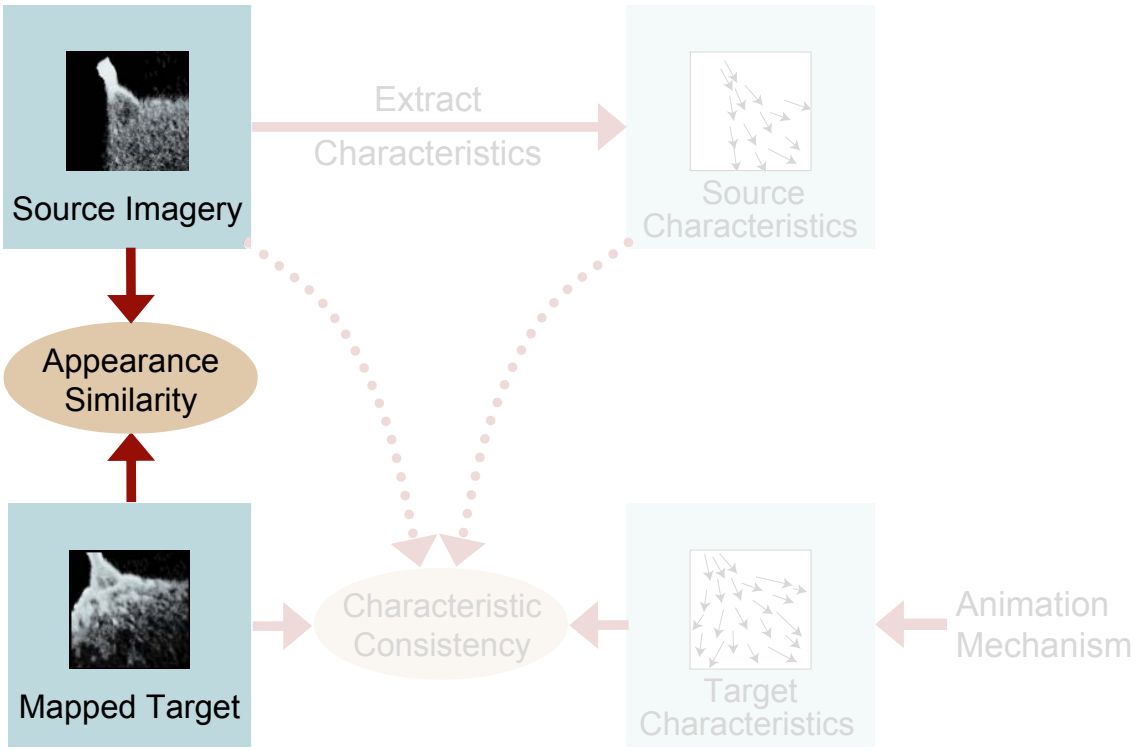


Figure 21: Specializing example-based rendering to only consider appearance similarity. The faded components of the schematic are ignored from the formulation. This reduces the problem to texture synthesis.

function is then optimized using an Expectation Maximization (EM)-like algorithm [40].

In contrast to most example-based techniques that do region-growing, this is a joint optimization approach that progressively refines the entire texture. This joint optimization results in progressive refinement of the entire texture as it is improved through successive iterations of our algorithm. This form of progressive refinement of textures would be very useful in situations that demand fast or real-time computations with level-of-details aspects like computer games. Additionally, this approach is ideally suited for extending the energy formulation to allow for controllable synthesis of textures.

5.1 Texture Optimization

To convert our probabilistically defined MRF prior on the texture into an energy function, we employ the following property of MRFs: the conditional probability density locally describing an MRF (such as $P(E_{\mathcal{T}}(p)|E_{\mathcal{T}}(\mathcal{N}_p \backslash p))$ in (15)), can be converted into a product of joint probability densities over these local variables. This is commonly known as the *Markov-Gibbs Equivalence* [37]. A Gibbs Random Field (GRF) is a set of random variables whose configurations obey the Gibbs distribution. A Gibbs distribution takes the following form:

$$P(\mathbf{x}) \propto \exp^{-U(\mathbf{x})},$$

where $U(\mathbf{x})$ is an energy function expressible as a sum of *clique potentials* over all possible cliques $\mathcal{C}$:

$$U(\mathbf{x}) = \sum_{c \in \mathcal{C}} V_c(\mathbf{x}).$$

For a given neighborhood system defined by $\mathcal{N}$ over an image lattice $\mathcal{S}$, cliques are subsets of pixels that are fully connected within that system, *i.e.*, all pixels in a clique are neighbors of each other. According to the Hammersley-Clifford theorem [27], an MRF on $\mathcal{S}$ with respect to $\mathcal{N}$ is also a GRF on $\mathcal{S}$ with respect to $\mathcal{N}$ and vice-versa. This means that an MRF-governed local probability density of the form $P(\mathbf{x}(p)|\mathbf{x}(\mathcal{N}_p \backslash p))$ can be equivalently written as a global GRF-governed probability density of the form:

$$\begin{aligned} P(\mathbf{x}) &\propto \exp^{-\sum_{c \in \mathcal{C}} V_c(\mathbf{x})} \\ &= \prod_{c \in \mathcal{C}} \exp^{-V_c(\mathbf{x})} \\ &= \prod_{c \in \mathcal{C}} P_c(\mathbf{x}) \end{aligned} \tag{17}$$

where $P(\mathbf{x})$ denotes the probability of observing the entire image $\mathbf{x}$, while $P_c(\mathbf{x})$ is the probability of a local clique of pixels. In what follows we will use an energy-based formulation of (17), *i.e.*, we will define an energy function over the image lattice expressed as a sum of

energies over local cliques:

$$U(\mathbf{x}) = \sum_{c \in C} V_c(\mathbf{x})$$

Also, from now onwards we will use the word *neighborhood* to actually mean a clique of pixels over which local energy measures are defined.

We now describe our *texture energy* metric that measures the similarity of the synthesized texture to the input sample. We define this energy in terms of the similarity of local neighborhoods in the texture to local neighborhoods in the input. We postulate that a *sufficient* condition for a texture to be similar to the input sample is that all neighborhoods in the texture are similar to some neighborhood in the input. This requires that the neighborhood size be large enough to capture the repeating elements in the texture. We define the energy of a single neighborhood to be *its distance to the closest neighborhood in the input*. The total energy of the texture is then equal to the sum of energies over individual local neighborhoods in the texture.

Formally, let X denote the texture over which we want to compute the texture energy and Z denote the input sample to be used as reference. Let $\mathbf{x}$ be the *vectorized* version of X, *i.e.,* it is formed by concatenating the intensity values of all pixels in X. For a pre-specified neighborhood width w, let $\mathcal{N}_p$ represent the neighborhood in X centered around pixel p. Then, the *sub-vector* of $\mathbf{x}$ that corresponds to the pixels in $\mathcal{N}_p$ is denoted by $\mathbf{x}_p$. Further, let $\mathbf{z}_p$ be the vectorized pixel neighborhood in Z that is closest to $\mathbf{x}_p$ under the Euclidean norm. Then, we define the texture energy over X to be

$$U_t(\mathbf{x}; \{\mathbf{z}_p\}) = \sum_{p \in X^\dagger} \|\mathbf{x}_p - \mathbf{z}_p\|^2. \tag{18}$$

This is shown schematically in Figure 22. We only consider neighborhoods centered around pixels in a set $X^\dagger \subset X$ for computing the energy. We do so because in practice, it is redundant and computationally expensive to compute the energy over all neighborhoods in the texture – the primary computational expense lies in the search for input neighborhoods $\mathbf{z}_p$. Therefore, we pick a subset of neighborhoods that sufficiently overlap with each other and

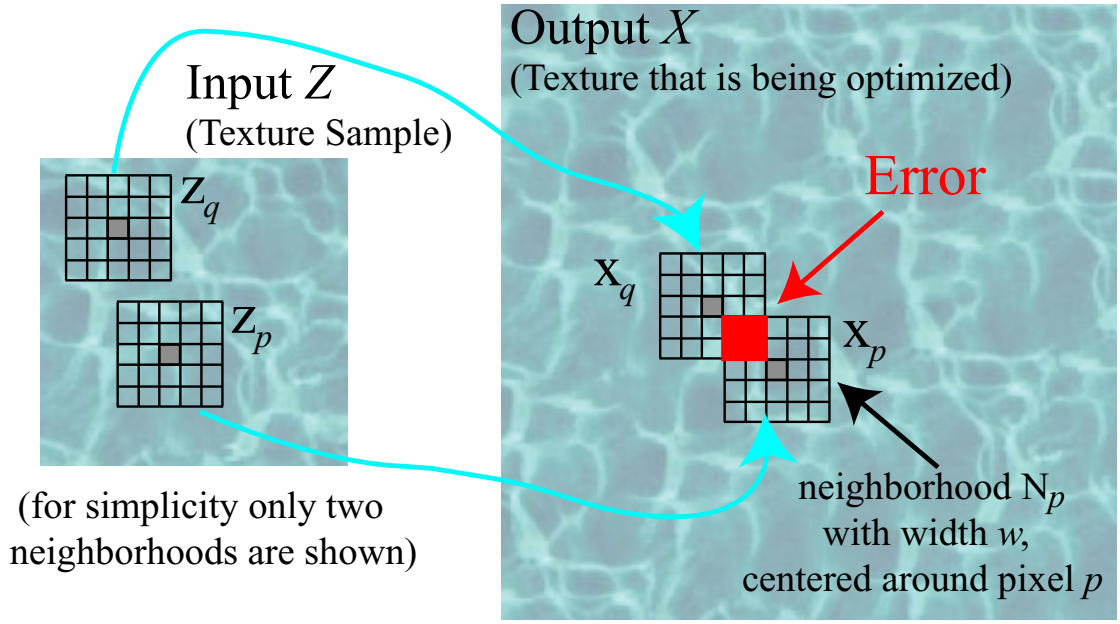


Figure 22: Schematic demonstrating our texture similarity metric. The energy of neighborhood $\mathbf{x}_p$ centered around pixel p is given by its distance to the closest input neighborhood $\mathbf{z}_p$. When two neighborhoods $\mathbf{x}_p$ and $\mathbf{x}_q$ overlap, then any mismatch between $\mathbf{z}_p$ and $\mathbf{z}_q$ will lead to accumulation of error in the overlapping region (shown in red).

define the energy only over this subset. We have chosen $X^\dagger$ to consist of neighborhood centers that are $w/4$ pixels apart, where w is the width of each neighborhood.

We can use the above formulation to perform texture synthesis by iteratively refining an initial estimate of the texture, decreasing the texture energy at each iteration. During each iteration, we alternate between $\mathbf{x}$ and $\{\mathbf{z}_p : p \in X^\dagger\}$ as the variables with respect to which (18) is minimized. Given an initialization of the texture, $\mathbf{x}$, we first find the closest input neighborhood $\mathbf{z}_p$ corresponding to each output neighborhood $\mathbf{x}_p$. We then update $\mathbf{x}$ to be the texture that minimizes the energy in (18) – note that we treat $\mathbf{x}$ as a real-valued continuous vector variable. Since $\mathbf{x}$ changes after this update step, the set of closest input neighborhoods $\{\mathbf{z}_p\}$ may also change. Hence, we need to repeat the two steps iteratively until convergence, *i.e.*, until the set $\{\mathbf{z}_p\}$ stops changing. If $\mathbf{x}$ is initially unknown, then we bootstrap the algorithm by assigning a random neighborhood from the input to each $\mathbf{z}_p$. Algorithm 1 describes the pseudocode for our texture synthesis algorithm.

Algorithm 1 Texture Synthesis

$\mathbf{z}_p^0 \leftarrow$ random neighborhood in $Z \ \ \forall p \in X^\dagger$
for iteration $n = 0 : N$ **do**
  $\mathbf{x}^{n+1} \leftarrow \arg\min_{\mathbf{x}} U_t(\mathbf{x}; \{\mathbf{z}_p^n\})$
  $\mathbf{z}_p^{n+1} \leftarrow$ nearest neighbor of $\mathbf{x}_p^{n+1}$ in $Z \ \ \forall p \in X^\dagger$
  if $\mathbf{z}_p^{n+1} = \mathbf{z}_p^n \ \ \forall p \in X^\dagger$ **then**
    $\mathbf{x} \leftarrow \mathbf{x}^{n+1}$
    break
  end if
end for

Our approach is algorithmically similar to Expectation-Maximization (EM) [40]. EM is used for optimization in circumstances where, in addition to the desired variables, the parameters of the energy function being optimized are also unknown. Therefore, one alternates between estimating the variables and the parameters in the E and the M steps respectively. In our case, the desired variable is the texture image, $\mathbf{x}$, while the parameters are the input neighborhoods, $\{\mathbf{z}_p\}$. The two steps of our algorithm can be thought of as E and M steps. The estimation of $\mathbf{x}$ by minimizing the texture energy in (18) corresponds to the E-step, while finding the set of closest input neighborhoods, $\{\mathbf{z}_p\}$, corresponds to the M-step.

In the E-step, we need to minimize (18) w.r.t. $\mathbf{x}$. This is done by setting the derivative of (18) w.r.t. $\mathbf{x}$ to zero, which yields a linear system of equations that can be solved for $\mathbf{x}$. To express the equation mathematically, we first rewrite (18) as

$$U_t(\mathbf{x}; \{\mathbf{z}_p\}) = \sum_{p \in X^\dagger} (\mathbf{W}_p \mathbf{x} - \mathbf{z}_p)^T (\mathbf{W}_p \mathbf{x} - \mathbf{z}_p)$$

where $\mathbf{W}_p$ is a *selection matrix* corresponding to neighborhood $\mathcal{N}_p$. This matrix selects the pixels that form the sub-vector $\mathbf{x}_p$ from the entire image vector $\mathbf{x}$. The structure of $\mathbf{W}_p$ is similar to a permutation matrix, except that it is rectangular. Each row of this matrix contains only a single non-zero entry with value '1'. If the i^{th} scalar in the sub-vector $\mathbf{x}_p$ corresponds to the j^{th} scalar in $\mathbf{x}$, then the entry $\mathbf{W}_p(i, j) = 1$. Having this selection matrix is convenient because it allows us to easily take derivatives w.r.t. $\mathbf{x}$. The linear system

consequently obtained is

$$\sum_{p \in X^\dagger} \mathbf{W}_p^T \mathbf{W}_p \mathbf{x} = \sum_{p \in X^\dagger} \mathbf{W}_p^T \mathbf{z}_p, \tag{19}$$

which is an equation of the form

$$\mathbf{M}\mathbf{x} = \mathbf{b}, \tag{20}$$

where $\mathbf{M}$ is a diagonal matrix and $\mathbf{b}$ is a vector whose value is determined by the set of input neighborhood sub-vectors $\{\mathbf{z}_p\}$ – these sub-vectors are known after the M-step. $\mathbf{b}$ is actually a sum over these input neighborhoods, where the neighborhoods are displaced to be centered over the appropriate pixel locations before summation. The LHS sums over unknown pixel values in $\mathbf{x}_p$. Note that since the same pixel may appear in multiple neighborhoods $\mathbf{x}_p$, the LHS will sum over the same pixel multiple times. The solution of this equation is to assign to each output pixel, the average value of input neighborhood pixels that correspond to that location. Note that for a quadratic $U_t(\mathbf{x}; \{\mathbf{z}_p\})$ (as in (18)), this minimization is equivalent to computing the expected value (or mean) of $\mathbf{x}$ under the following probability distribution[1]:

$$p(\mathbf{x}; \{\mathbf{z}_p\}) \propto \exp(-U_t(\mathbf{x}; \{\mathbf{z}_p\})).$$

The M-step of our algorithm minimizes (18) with respect to the set of input neighborhoods, $\{\mathbf{z}_p\}$, keeping $\mathbf{x}$ fixed at the value estimated in the E-step. This requires us to solve a nearest neighbor search problem: for each $\mathbf{x}_p$, we need to find its nearest neighbor $\mathbf{z}_p$ from the input. To accelerate this search, we use hierarchical clustering to organize the input neighborhoods into a tree structure [31, 19, 15]. Starting at the root node, we perform *k-means* clustering (with $k = 4$) over all input neighborhoods contained in that node. We then create k children nodes corresponding to the k clusters and recursively build the tree for each of these children nodes. The recursion stops when the number of neighborhoods

[1]Typically, in order to perform exact EM, the E-step should also compute the covariance of $\mathbf{x}$ in addition to its mean. Our formulation, on the other hand, only computes the mean as it is based on an energy function and not a probability distribution. Even then, we end up performing exact EM, because the covariance does not affect the outcome of the M-step in our case.

in a node falls below a threshold (1% of total in our implementation). In order to handle large neighborhood sizes, we employ a memory-efficient adaptation that does not explicitly store neighborhood sub-vectors at leaf nodes. Instead, it records just the neighborhood's location in the input image and the corresponding sub-vectors are constructed on the fly, as necessary.

In order to visualize the optimization process, one can think of each term in (18) as the potential resulting from a force that pulls the pixels in $\mathbf{x}_p$ towards pixels in $\mathbf{z}_p$. Minimization of this potential corresponds to bringing each sub-vector $\mathbf{x}_p$ as close to $\mathbf{z}_p$ as possible. If neighborhoods centered around different pixels p and q overlap with each other, then the corresponding sub-vectors $\mathbf{x}_p$ and $\mathbf{x}_q$ will contain pixels that are common with each other. Each such common pixel is pulled towards possibly different intensity values by $\mathbf{z}_p$ and $\mathbf{z}_q$. The outcome of the minimization procedure is to assign an intensity value to the common pixel that is equal to the average of the corresponding values in $\mathbf{z}_p$ and $\mathbf{z}_q$.

Intuitively, our algorithm tries to find *good* relative arrangements of input neighborhoods in order to synthesize a new texture. During each iteration, the M-step chooses an arrangement of input neighborhoods that best explains the current estimate of the texture. The averaging in the E-step allows overlapping neighborhoods to communicate consistency information among each other: neighborhoods that don't match well with each other cause blurring in the synthesized texture. This blurred region represents a transition between two inconsistent regions and may be significantly different in appearance from the input neighborhoods that determine it. This allows the next iteration of the M-step to replace the input neighborhoods corresponding to this region with ones that are more consistent with each other, *i.e.*, neighborhoods that carry out the transition but get rid of the blurring.

5.2 Robust Formulation

The texture energy as defined in (18) performs least squares estimation of $\mathbf{x}$ w.r.t. $\mathbf{z}_p$. This causes outliers – $\mathbf{z}_p$ that are not very close to $\mathbf{x}_p$ – to have an undue influence on $\mathbf{x}$. Also,

it is desirable to not change $\mathbf{x}_p$ by much if it is already very close to $\mathbf{z}_p$. This can be accomplished by using a robust energy function: we replace the squared term $\|\mathbf{x}_p - \mathbf{z}_p\|^2$ in (18) with $\|\mathbf{x}_p - \mathbf{z}_p\|^r$, where $r < 2$. This energy function belongs to a class of robust regressors, called M-estimators, that are typically solved using iteratively re-weighted least squares (IRLS) [12]. IRLS is an iterative technique in which a weighted least squares problem is solved during every iteration. The weights are adjusted at the end of the iteration, and this procedure is repeated until convergence. Our synthesis algorithm naturally lends itself to IRLS: before applying the E-step, we choose, for each neighborhood $\mathcal{N}_p$, a weight $\omega_p = \|\mathbf{x}_p - \mathbf{z}_p\|^{r-2}$ – in our implementation, we have used $r = 0.8$. We then minimize the modified energy function:

$$U_t(\mathbf{x}; \{\mathbf{z}_p\}) = \sum_{p \in X^\dagger} \omega_p \|\mathbf{x}_p - \mathbf{z}_p\|^2.$$

Additionally, we apply a per pixel weight within the energy term for each neighborhood, so that pixels closer to the center of the neighborhood have a greater bearing on the texture than those far away. Specifically, we use a Gaussian fall-off function that smoothly decreases the pixel weight as it moves away from the neighborhood center (Efros et al. [17] use a similar weighting scheme for nearest-neighbor search). This yields the following energy function:

$$U_t(\mathbf{x}; \{\mathbf{z}_p\}) = \sum_{p \in X^\dagger} \omega_p (\mathbf{x}_p - \mathbf{z}_p)^T \mathbf{G} (\mathbf{x}_p - \mathbf{z}_p),$$

where $\mathbf{G}$ is a diagonal weight matrix corresponding to the Gaussian fall-off function. Minimizing this energy requires solving an equation that is only slightly different from (19): it now contains a *weighted* sum over pixels on both RHS and LHS.

5.3 Gradient-based Energy

We can generalize the energy function defined in (18) to incorporate other characteristics of the texture besides color. For example, in order to use image gradients as an additional

similarity metric, we define the energy as

$$U_t(\mathbf{x};\{\mathbf{z}_p\}) = \sum_{p\in X^\dagger} \|\mathbf{x}_p - \mathbf{z}_p\|^2 + \mu \sum_{p\in X^\dagger} \|\mathbf{D}\mathbf{x}_p - \mathbf{D}\mathbf{z}_p\|^2, \tag{21}$$

where $\mathbf{D}$ is the differentiation operator and μ is a relative weighting coefficient. Minimizing this function w.r.t. $\mathbf{x}$ yields a linear system that is not diagonal – as was the case in (19) – but is still sparse and hence can be solved efficiently. To obtain this system, we will again make use of the selection matrix $\mathbf{W}_p$. We rewrite (21) as

$$U_t(\mathbf{x};\{\mathbf{z}_p\}) = \sum_{p\in X^\dagger} \|\mathbf{W}_p\mathbf{x} - \mathbf{z}_p\|^2 + \mu \sum_{p\in X^\dagger} \|\mathbf{D}\mathbf{W}_p\mathbf{x} - \mathbf{D}\mathbf{z}_p\|^2.$$

Differentiating this U_t w.r.t. $\mathbf{x}$ and setting it to zero then yields the following equation:

$$\sum_{p\in X^\dagger} \mathbf{W}_p^T\mathbf{W}_p\mathbf{x} + \mu \sum_{p\in X^\dagger} \mathbf{W}_p^T\mathbf{D}^T\mathbf{D}\mathbf{W}_p\mathbf{x} = \sum_{p\in X^\dagger} \mathbf{W}_p^T\mathbf{z}_p + \mu \sum_{p\in X^\dagger} \mathbf{W}_p^T\mathbf{D}^T\mathbf{D}\mathbf{z}_p. \tag{22}$$

This equation is still linear and has a form similar to (20):

$$\mathbf{M}^*\mathbf{x} = \mathbf{b}^*,$$

where

$$\begin{aligned}\mathbf{M}^* &= \sum_{p\in X^\dagger} \mathbf{W}_p^T\mathbf{W}_p + \mu \sum_{p\in X^\dagger} \mathbf{W}_p^T\mathbf{D}^T\mathbf{D}\mathbf{W}_p \\ \mathbf{b}^* &= \sum_{p\in X^\dagger} \mathbf{W}_p^T\mathbf{z}_p + \mu \sum_{p\in X^\dagger} \mathbf{W}_p^T\mathbf{D}^T\mathbf{D}\mathbf{z}_p.\end{aligned}$$

In (22), the first term on the LHS as well as RHS is the same as in (19) – these are $\mathbf{M}$ and $\mathbf{b}$ respectively. The remaining terms take the gradient information into account. It turns out that the matrix operator in the second term on the LHS (the second part of the sum forming $\mathbf{M}^*$) simplifies to a form of weighted Laplacian operator applied to the entire image vector $\mathbf{x}$. This is not surprising because the Laplacian operator is defined as $\mathbf{D}^T\mathbf{D}$ when the differentiation operator $\mathbf{D}$ applies to the whole image. However, in this case, $\mathbf{D}$ is defined only over a single neighborhood. Nevertheless, the weighted summations over these neighborhood operators eventually lead to a matrix that has the same form as the

Laplacian operator applied to the whole image. Thus, the system matrix is a sparse banded positive-definite matrix and therefore can be solved for efficiently. In our implementation, we have solved this equation using pre-conditioned conjugate gradients [54].

Note that even though we have experimented with color and gradient, one could use other energy functions of the form $\|\psi(\mathbf{x}_p) - \psi(\mathbf{z}_p)\|^2$ where $\psi(\mathbf{x}_p)$ measures some property of the texture neighborhood $\mathbf{x}_p$. The only requirement is that we should be able to optimize ψ w.r.t. $\mathbf{x}$.

5.4 Multi-level Synthesis

We use our algorithm in a multi-resolution and multi-scale fashion. We first synthesize our texture at a coarse resolution, and then up-sample it to a higher resolution via interpolation. This serves as the initialization of the texture at the higher resolution. Also, within each resolution level, we run our synthesis algorithm using multiple neighborhood sizes in order from largest to smallest. We start with the largest neighborhood size in order to align the large scale structures of the texture first. Subsequently, we synthesize with smaller neighborhood sizes (in decreasing order) to eliminate fine scale errors from the synthesized texture. We term the synthesis with different neighborhood sizes as synthesis at multiple scale levels. This is reasonable since each neighborhood size respects texture structures at a different scale. The synthesized result at each scale level serves as an initialization for the synthesis at the next level.

Such a multi-level approach is helpful because it allows the finer scale synthesis to begin with a good initialization of the texture, thereby avoiding undesirable local minima. Intuitively, at a lower resolution, texture neighborhoods are spatially close to each other and it is easier to propagate consistency information across the entire texture. In our experiments, we generally use three resolution levels and successive neighborhood sizes of 32×32, 16×16, and 8×8 pixels – at each resolution, only those neighborhood sizes are used that fit in the the corresponding (potentially sub-sampled) input image.

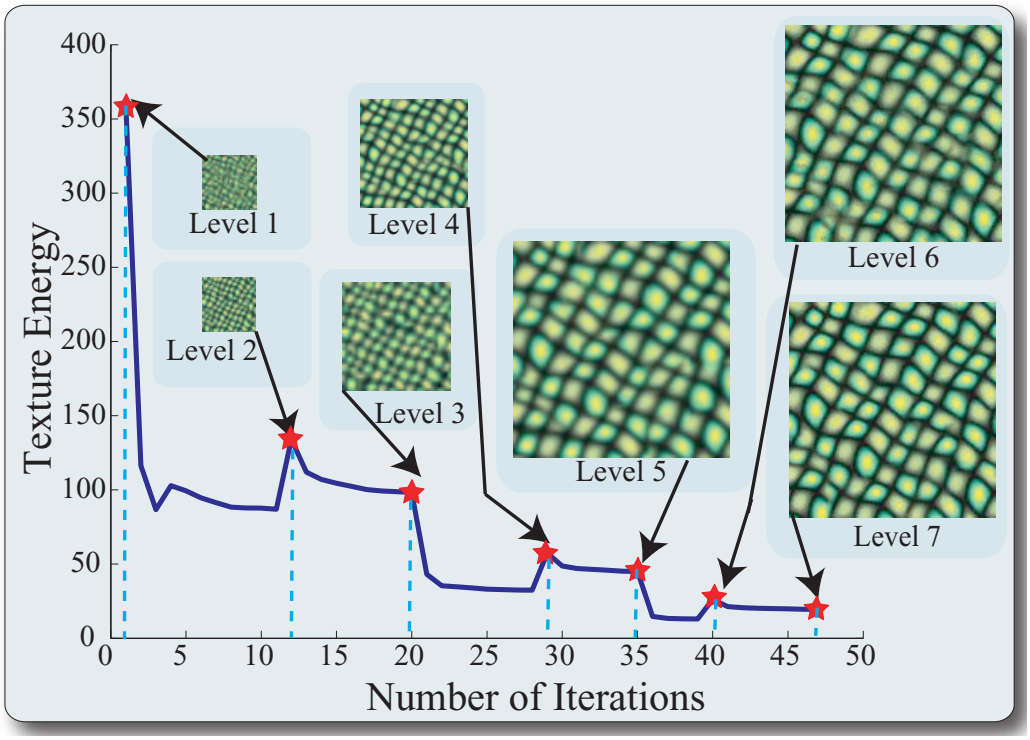


Figure 23: Texture energy plotted as a function of number of iterations. Also shown is the synthesized texture after each resolution and scale (neighborhood size) level. Level 1 shows the random initialization. Level 2 shows synthesis at (1/4 resolution, 8×8 neighborhood), Level 3: (1/2, 16×16), Level 4: (1/2, 8×8), Level 5: (1, 32×32), Level 6: (1, 16×16), Level 7: (1, 8×8).

In Figure 23, we plot the energy of the synthesized texture as a function of number of iterations. The iterations for various resolution and scale levels are concatenated in the order of synthesis – we normalize the energy at each level by the number of pixels and neighborhood size. We also show the synthesized texture at the end of each synthesis level. The texture energy generally decreases as the number of iterations increase, thereby improving texture quality. Note that intermediate iterations produce textures that appear to be coarse-scale approximations of the final texture. One can potentially exploit this progressive refinement property to synthesize textures in a level-of-detail fashion. The jumps seen in the energy plot are due to change of synthesis level – resolution or neighborhood size. They are in fact desirable because they help the algorithm to get out of poor local minima.

5.5 Results

We have applied our algorithm for synthesizing both image and video textures. In the case of image textures, we perform multi-resolution synthesis with three resolution levels (each coarse level has half the resolution of the finer level). Also, within each resolution level, we perform synthesis using multiple neighborhood sizes. We have generally used neighborhood sizes of 32×32, 16×16, and 8×8 pixels in our results. Figure 24 shows some of the results for image texture synthesis. It works well for a wide range of textures varying from stochastic to structured. In Figure 25, we show comparisons with other techniques. The results of our technique are generally at par with the the state-of-the-art in texture synthesis. In fact, synthesis using texture optimization tends to be less repetitive than synthesis using graph cuts, as shown in the text example (bottom row) in Figure 25.

In order to synthesize video textures, the only change we need to make is to increase the dimensionality of the domain by one, *i.e.*, we consider 3D neighborhoods in space-time instead of 2D neighborhoods. The entire spatio-temporal volume of the video is synthesized at once. We perform multi-resolution synthesis in the case of video as well. However, we do not sub-sample along the temporal dimension. This is because the temporal dimension is sparse in terms of available data to begin with – there are usually not enough frames to sub-sample. Also, the dynamics of the video sequence along time are different from its variations along space, and perceptually, it is desirable to give greater attention to even fine-scale changes along time. In the case of video, we have used three resolution levels but only one neighborhood size at each level: $16 \times 16 \times 4$. Figure 26 shows stills from example videos that we have used for video texture synthesis. Our results for video textures using texture optimization are comparable to those obtained using graph cuts. However, intermittent blurring is observable at times, since the optimization approach involves blending across neighborhoods.

The computational complexity of our technique is dominated by nearest neighbor search (M-step). It is linear in the number of nearest neighbor calls, which are $O(\frac{n_o}{w^2})$ per

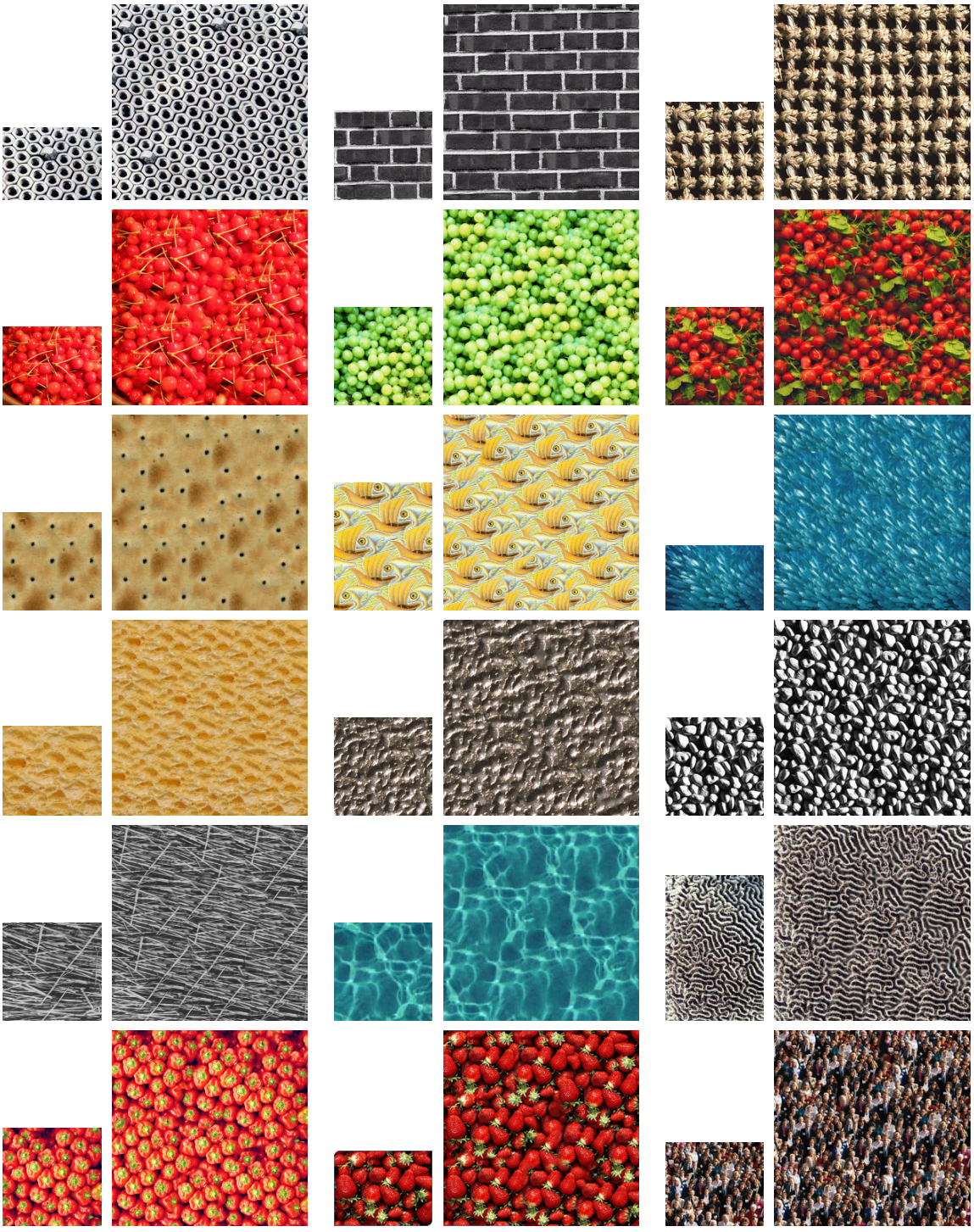


Figure 24: Results for image texture synthesis. For each texture, the input is on the left and the output on the right.

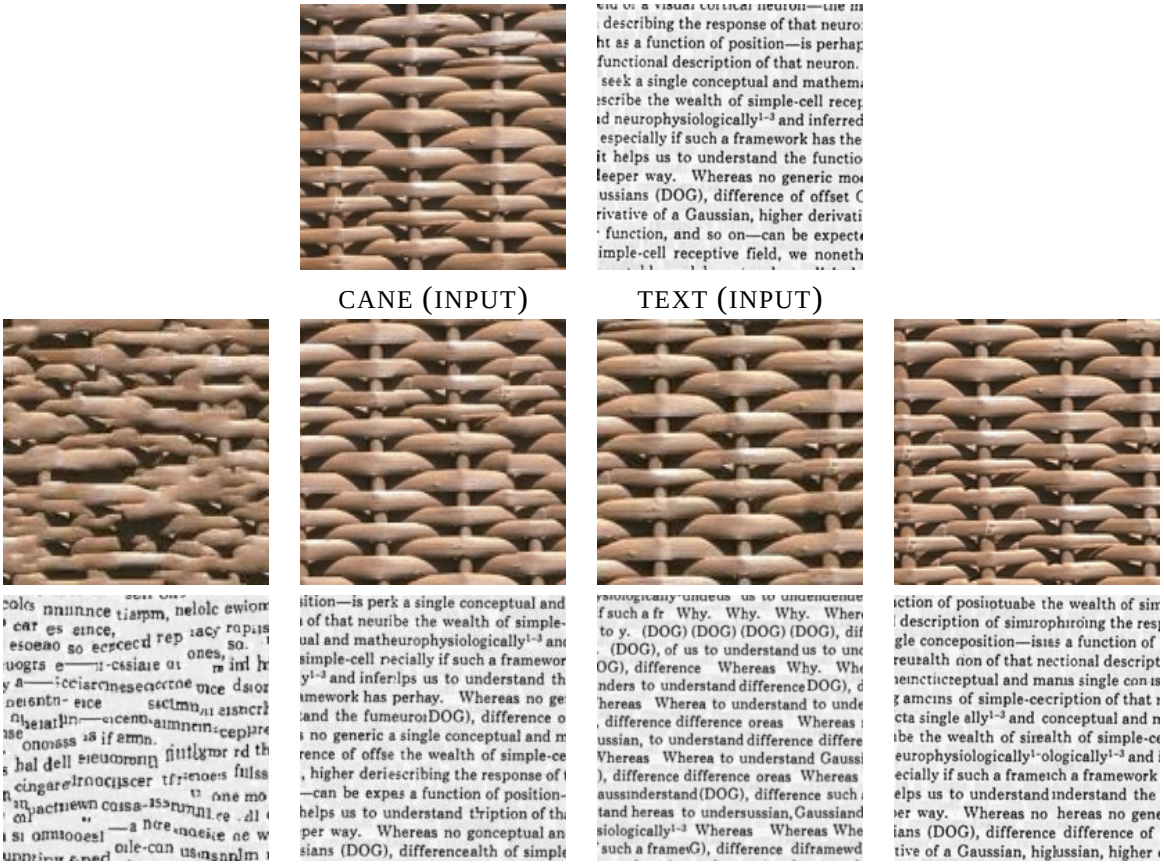


Figure 25: Comparison with various other texture synthesis techniques. The input textures are shown in the top row. The bottom two rows show the comparison.

Figure 26: Examples of videos textures that were synthesized using texture optimization.

iteration – n_o is the number of pixels in the output texture while w is the width of the neighborhood. Theoretically, the time taken per call is $O(w^2)$. In practice, however, we found it to be less than that – the exact dependence is not known to us as it is governed by MATLAB's vectorized matrix multiplication algorithm. The implication is that execution is faster for larger w. Actual execution time per iteration at different resolution levels (averaged over the various neighborhood sizes used within that resolution) was: 2.5 seconds at 64×64, 10 seconds at 128×128, and 25 seconds at 256×256. Usually 3-5 iterations per resolution/scale level were found to be enough. Average total execution time for multi-level synthesis of 256×256 textures was 7-10 minutes, while for 128×128 textures, it was 1-3 minutes. For video textures, it took 15-90 minutes for synthesizing 100×100 videos of varying temporal length. All timing results are reported for our unoptimized MATLAB code, running on a dual-processor 2.4GHz PC with 2GB RAM.

5.5.1 Discussion

Our results for both image and video textures are comparable to the state-of-the-art. Our synthesis algorithm iteratively improves texture quality and is therefore suitable for progressive refinement of the synthesized texture – this may allow it to be used in conjunction with level-of-detail applications, *e.g.*, video games. However, since it tries to decrease the energy at each iteration, it can get stuck in local minima. This usually happens because neighborhoods that are located far from each other can communicate only through intermediate overlapping neighborhoods. Multi-resolution synthesis is important in this context because it brings the neighborhoods closer location-wise.

Another property of our synthesis technique is that it can be thought of as a projection technique, as discussed in Section 5.1. Starting with an initial estimate of the texture, it brings it as close as possible to the input texture in terms of local similarity. Figure 27 shows the results of an experiment that we conducted to test this property. We took a checkerboard pattern and applied a deformation to it using the *twirl* operator in Adobe Photoshop. The original texture pattern is shown at the top. The second row shows the deformed texture, obtained after applying different degrees of twirl to it. We applied our texture optimization technique to correct these deformed textures – using them as the initial texture – to synthesize the results shown in the last two rows. Of these, the third row shows results obtained after applying the synthesis algorithm at a single (finest) resolution only. Our technique is able to correct the deformed texture and bring it close to the original for the first two deformations. However, in the most extreme case (third column), it gets trapped in a local minima leading to a less than satisfactory result. The fourth row shows the result of applying multi-resolution synthesis to the deformed textures. Here, we used sub-sampled versions of the deformed textures to initialize the synthesis process. As demonstrated by the results, multi-resolution synthesis has less tendency of settling for local minima.

The texture energy as defined in this chapter only looks at the color and gradient of the texture. For future work, we would like to experiment with the use of other texture

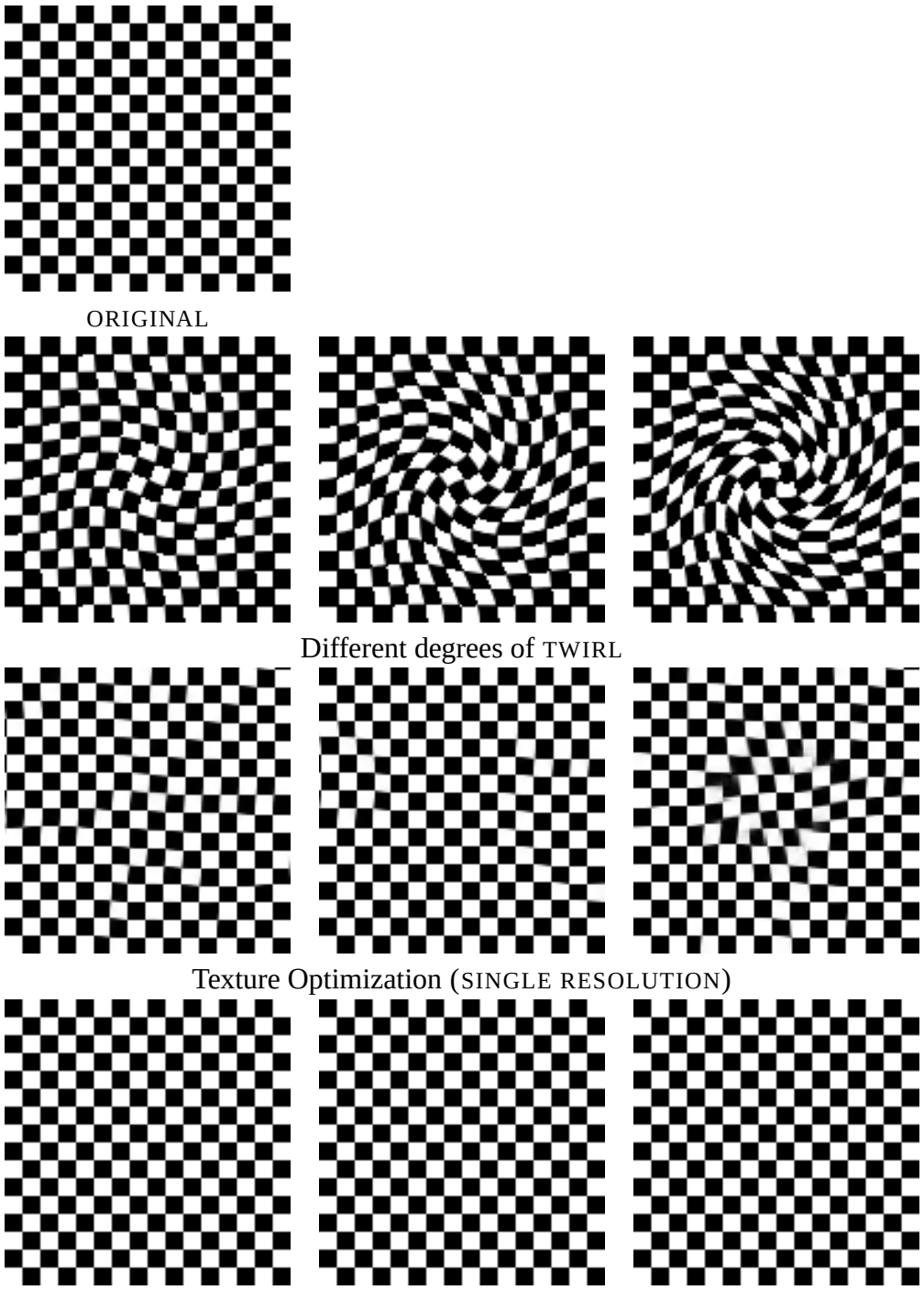


Figure 27: Correcting a deformed texture using Texture Optimization. The original texture (top) is deformed by applying different degrees of twirl to it (shown in second row). The third row shows texture optimization applied to these deformed textures at only the finest resolution. The fourth row shows results for multi-resolution synthesis.

properties such as filter outputs or edge information in designing the energy function.

5.6 Summary

In this chapter, a novel technique for texture synthesis using optimization has been presented. It results from specializing the probabilistic formulation presented in the last chapter, so that only appearance similarity between the synthesized and input textures is considered. We then transform the probability maximization problem to an energy minimization problem and optimize for it using an iterative EM-like algorithm. The energy-based formulation provides a quality of the synthesized texture at each iteration of the optimization process and continually improves it. Previous approaches for texture synthesis generally lack any such quantitative estimates of the quality of the synthesized texture. The results obtained using this technique are comparable to the state-of-the-art, including the graph cuts approach presented in Chapter 3. As will be shown in the next chapter, the energy-based formulation presented here can also be extended to incorporate additional terms that specify desirable properties or characteristics of the synthesized texture. These terms allow us to add controllability to the synthesis process. In that sense, the texture optimization approach is superior to the graph-cuts approach, even if the quality of its results is not necessarily better than that of graph-cuts. The graph-cuts technique can synthesize a wide variety of image and video textures with extremely realistic results. However, controlling it is difficult because of the fact that it copies large patches to the output – a single patch placement step can affect large portions of the texture. Texture optimization, on the other hand, operates at multiple resolution and scale levels which allows a finer level of control over the synthesis process.

CHAPTER VI

CONTROLLABLE SYNTHESIS: FLOWING IMAGE TEXTURES

Until now, we have discussed two techniques for texture synthesis – one based on graph-cuts and the other based on optimization – and also presented a framework within which algorithms for example-based rendering can be designed. The last chapter described an optimization technique for texture synthesis that was obtained by specializing the example-based rendering framework to only consider appearance similarity. In this chapter, we consider both the appearance similarity and the characteristic consistency aspects of our framework. We again transform the probabilistic formulation of Chapter 4 into an energy minimization problem as was done in the last chapter. In fact, the new formulation is a simple extension of the formulation presented in the previous chapter. It accommodates characteristic consistency by adding an additional energy term that act as a control term for the specific characteristic of interest.

Besides discussing the extension of texture optimization for controllable synthesis in general, we also describe a specific instance of example-based rendering in this chapter. We consider the case in which the source imagery is an image texture while the characteristic being controlled is image motion, represented as a flow-field. We have implemented a system that generates *texture sequences* that follow a given flow-field while maintaining appearance similarity to a given source texture. This means that the shape, size, orientation, etc. of the texture elements is maintained even as they follow the given flow-field. The flow-field must be a two-dimensional vector field but it is allowed to dynamically change over time. In the next chapter, we will discuss the case in which the source imagery is a video

texture instead of an image texture, while the characteristic is still flow.

The motivation for using flow-fields as the control mechanism is three-fold. *Firstly*, flow is directly related to the *motion* in the scene. Motion provides a very important cue in scene understanding. This also makes it a desirable quantity to control in a synthetic scene. Even in the context of video editing, where the goal may be to modify a real scene, motion can be used to characterize the editing operation. For example, if an editor wants to add an obstacle in the path of a stream, then one can describe this operation by the change in the stream's flow that would occur after adding the obstacle. *Secondly*, many textures like water, fire, smoke, etc are visual manifestations of phenomena that can be physically described as fluid flow. This opens up the opportunity of using example-based rendering in conjunction with fluid simulation techniques as they usually generate a flow-field as output. *Finally*, the technique developed in this chapter can be used as a method for flow visualization, since the output sequence animates a texture as guided by the given flow-field. The significance of our technique is that it allows for a rich variety of textures to be used for visualizing the flow-field.

6.1 Controllable Synthesis

To perform controllable synthesis within our framework, we need to consider characteristic consistency in addition to appearance similarity. Recall that our probabilistic formulation treats these terms as likelihood and prior respectively. We want to maximize the posterior which is decomposed as the product of likelihood and prior. This was expressed mathematically in (10):

$$\widehat{E}_{\mathcal{T}} = \arg\max_{E_{\mathcal{T}}} P(\mathbf{C}_{\mathcal{T}}|E_{\mathcal{T}},E_{\mathcal{S}},\mathbf{C}_{\mathcal{S}}).P(E_{\mathcal{T}}|E_{\mathcal{S}}).$$

The first term in the product is the likelihood while the second term is the prior. We transform this probability maximization problem into an energy minimization problem as was done in the last chapter:

$$\widehat{E}_{\mathcal{T}} = \arg\min_{E_{\mathcal{T}}} [-\log P(\mathbf{C}_{\mathcal{T}}|E_{\mathcal{T}},E_{\mathcal{S}},\mathbf{C}_{\mathcal{S}}) - \log P(E_{\mathcal{T}}|E_{\mathcal{S}})].$$

Here, $-\log P(E_{\mathcal{T}}|E_{\mathcal{S}})$ is the appearance or texture similarity term that measures texture energy. On the other hand, $-\log P(\mathbf{C}_{\mathcal{T}}|E_{\mathcal{T}},E_{\mathcal{S}},\mathbf{C}_{\mathcal{S}})$ represents the characteristic consistency term. This term measures how well the synthesized texture (sequence) matches with the characteristics. In the following discussion, we will also refer to the characteristic as the control criteria, and to its energy term as the control energy. We will make a switch in the notation to be consistent with the discussion in the previous chapter. The texture being synthesized, $E_{\mathcal{T}}$, will be denoted by $\mathbf{x}$ or X as appropriate. The source texture $E_{\mathcal{S}}$ will be denoted by $\mathbf{z}$ or Z as appropriate. To simplify notation, we will use $\mathbf{u}$ to denote all the control parameters which may include a subset of $\mathbf{C}_{\mathcal{T}}$, $\mathbf{C}_{\mathcal{T}}$, and $E_{\mathcal{S}}$, and any other associated variables. The various energy terms that will be used are defined as follows:

$$\begin{aligned} U_t(\mathbf{x};Z) &= -\log P(E_{\mathcal{T}}|E_{\mathcal{S}}) \\ U_c(\mathbf{x};\mathbf{u}) &= -\log P(\mathbf{C}_{\mathcal{T}}|E_{\mathcal{T}},E_{\mathcal{S}},\mathbf{C}_{\mathcal{S}}) \\ U(\mathbf{x}) &= U_t(\mathbf{x};Z)+\lambda U_c(\mathbf{x};\mathbf{u}). \end{aligned}$$

Here, $U_t(\mathbf{x};Z)$ denotes the texture energy, $U_c(\mathbf{x};\mathbf{u})$ denotes control energy, and $U(\mathbf{x})$ denotes the total energy that we wish to minimize. The coefficient λ is used to weight the two terms relative to each other. The control term, $U_c(\mathbf{x};\mathbf{u})$, attempts to satisfy the control criteria in the synthesized texture, while the texture term, $U_t(\mathbf{x};Z)$, tries to ensure that it is a representative sample of the input texture.

6.1.1 Adapting Texture Optimization for Controllability

The total energy that we need to minimize for controllable synthesis is the sum of texture and control energy terms. This means that the texture energy can be defined independently of the control energy. Hence, we can use the formulation from the previous chapter to define texture energy:

$$U_t(\mathbf{x};Z) = U_t(\mathbf{x};\{\mathbf{z}_p\}),$$

where $U_t(\mathbf{x};\{\mathbf{z}_p\})$ is as defined in (18). Recall that, in the last chapter, we defined the texture energy of the synthesized texture with respect to the source by adding up the squared

color differences between each synthesized texture neighborhood and its closest source texture neighborhood – $\{\mathbf{z}_p\}$ is the set of these closest source neighborhoods. To extend the optimization technique presented in the previous chapter to handle the control energy term, we simply minimize the total energy $U(\mathbf{x})$ at each iteration instead of just minimizing the texture energy $U_t(\mathbf{x};\{\mathbf{z}_p\})$.

In order to understand the addition of control criteria to the optimization process more clearly, lets consider a simple example. Lets say we want to perform *soft*-constrained synthesis where the desirable color values at certain pixel locations of the synthesized texture are specified. We can express the energy representing this criterion as the sum of squared distances between the synthesized and specified pixel values:

$$U_c(\mathbf{x};\mathbf{x}^c) = \sum_{k\in\mathcal{C}} (\mathbf{x}(k) - \mathbf{x}^c(k))^2, \tag{23}$$

where $\mathcal{C}$ is the set of constrained pixels and $\mathbf{x}^c$ is a vector containing the specified color values. While we want the pixels to be close to the desirable values at specified locations, we also want the overall texture appearance to be similar to the source texture. The two criteria are together expressed as the total energy that needs to be minimized:

$$U(\mathbf{x}) = U_t(\mathbf{x};\{\mathbf{z}_p\}) + \lambda U_c(\mathbf{x};\mathbf{x}^c). \tag{24}$$

Using (23) for the control term, $U_c(\mathbf{x};\mathbf{x}^c)$, and the definition of $U_t(\mathbf{x};\{\mathbf{z}_p\})$ from (18), we can rewrite the above equation as

$$U(\mathbf{x}) = \sum_{p\in X^\dagger} \|\mathbf{x}_p - \mathbf{z}_p\|^2 + \lambda \sum_{k\in\mathcal{C}} (\mathbf{x}(k) - \mathbf{x}^c(k))^2. \tag{25}$$

The minimization of $U(\mathbf{x})$ is done in a similar fashion as that of $U_t(\mathbf{x})$. We modify the E and M steps to account for $U_c(\mathbf{x})$ as follows.

In the E-step, we solve a new system of linear equations that results from the differentiation of $U(\mathbf{x})$ w.r.t. $\mathbf{x}$. For the constrained synthesis example described above, this corresponds to taking a weighted average of the specified pixel values and those corresponding to the output of texture synthesis at constrained locations. Mathematically, we

need to solve the following equation – obtained through differentiation of (25) w.r.t. $\mathbf{x}$:

$$\mathbf{M}\mathbf{x} + \lambda\mathbf{C}\mathbf{x} = \mathbf{b} + \lambda\mathbf{x}^c.$$

Here, $\mathbf{M}$ and $\mathbf{b}$ are the same as in (20). $\mathbf{C}$ is the matrix corresponding to the control term. It is a diagonal matrix with '1' as its diagonal entries wherever a constraint is provided, and '0' elsewhere, *i.e.,*

$$\mathbf{C}(k,l) = \begin{cases} 1 & k = l \text{ and } k \in \mathcal{C} \\ 0 & \text{otherwise} \end{cases}$$

Further, $\mathbf{x}^c$ is also taken to be non-zero only at the constraints, *i.e.,* $\mathbf{x}^c(k) = 0 \ \forall k \notin \mathcal{C}$. Note that the solution just described is specific to the soft-constraints control criterion. One can use more general energy functions as long as they can be optimized w.r.t. $\mathbf{x}$. Energy functions that lead to a linear system of equations in the E-step have the following general form:

$$U_c(\mathbf{x};\mathbf{u}) = (\mathbf{x}-\mathbf{c})^T\mathbf{F}(\mathbf{x}-\mathbf{c}),$$

where $\mathbf{F}$ is a symmetric positive semi-definite matrix and $\mathbf{c}$ is an arbitrary vector. The control vector $\mathbf{u}$ is specific to the criteria being considered and determines $\mathbf{F}$ and $\mathbf{c}$. Such quadratic energy functions are desirable because they can be minimized using linear optimization within each iteration of our texture synthesis algorithm.

We also modify the M-step, in which we search for the set of input neighborhoods $\{\mathbf{z}_p\}$. Even though $U_c(\mathbf{x})$ does not directly depend on $\{\mathbf{z}_p\}$, they are indirectly related with each other through $\mathbf{x}$. We exploit the fact that each synthesized neighborhood $\mathbf{x}_p$ will be similar to $\mathbf{z}_p$ after the E-step. Hence, when searching for $\mathbf{z}_p$, we look for input neighborhoods that are already consistent with the control criteria. The intuition is that if $\mathbf{z}_p$ has low control energy, U_c, then so will $\mathbf{x}_p$. For *each* $\mathbf{x}_p$, we find the $\mathbf{z}_p$ that minimizes the total energy, or equivalently the part of it that is affected by $\mathbf{z}_p$:

$$U_p(\mathbf{x};\mathbf{z}_p) = \|\mathbf{x}_p - \mathbf{z}_p\|^2 + \lambda U_c(\mathbf{y};\mathbf{u}), \tag{26}$$

where, $\mathbf{y}$ is constructed from $\mathbf{x}$ by replacing its pixels in the neighborhood $\mathcal{N}_p$ with $\mathbf{z}_p$, *i.e.*,

$$\mathbf{y}(q) = \begin{cases} \mathbf{z}_p(q-p+w/2) & q \in \mathcal{N}_p \\ \mathbf{x}(q) & \text{otherwise} \end{cases}$$

Here, $w = (w_x, w_y)$ encodes the neighborhood width in both spatial dimensions. It is important to note that (26) contains only that part of the total energy that is affected by $\mathbf{z}_p$ – the closest input sub-vector corresponding to $\mathcal{N}_p$. Consequently, the summation sign in the texture energy term is missing and $\mathbf{y}$ is computed by modifying $\mathbf{x}$ only within the neighborhood $\mathcal{N}_p$. As an example, in the case of soft-constrained synthesis, the new M-step searches for $\mathbf{z}_p$ that minimizes $\|\mathbf{x}_p - \mathbf{z}_p\|^2 + \lambda\|\mathbf{z}_p - \mathbf{x}_p^c\|^2$, *i.e.*, an input neighborhood whose pixel values at the constrained locations are already close to the specified ones. In our implementation, this modified search is approximate because it is still done using a hierarchical search tree as described in the previous chapter. Algorithm 2 describes controllable texture synthesis in pseudocode for general control criteria.

Algorithm 2 Controllable Texture Synthesis

$\mathbf{z}_p^0 \leftarrow$ random neighborhood in $Z \;\; \forall p \in X^\dagger$
for iteration $n = 0 : N$ **do**
  $\mathbf{x}^{n+1} \leftarrow \arg\min_{\mathbf{x}} [U_t(\mathbf{x}; \{\mathbf{z}_p^n\}) + \lambda U_c(\mathbf{x}; \mathbf{u})]$
  $\mathbf{z}_p^{n+1} \leftarrow \arg\min_{\mathbf{v}} [\|\mathbf{x}_p - \mathbf{v}\|^2 + \lambda U_c(\mathbf{y}; \mathbf{u})]$, $\mathbf{v}$ is a neighborhood in Z and $\mathbf{y}$ is the same as $\mathbf{x}$ except for neighborhood $\mathbf{x}_p$ which is replaced with $\mathbf{v}$
  if $\mathbf{z}_p^{n+1} = \mathbf{z}_p^n \;\; \forall p \in X^\dagger$ **then**
    $\mathbf{x} \leftarrow \mathbf{x}^{n+1}$
    break
  end if
end for

6.2 Flow-guided Synthesis using Image Textures

In this section, we consider a specific instantiation of control of characteristics of the synthesized texture. This is a special case of the example-based rendering framework presented in Chapter 4 in which an image texture is used as the source imagery while flow is

the characteristic that is controlled. The goal is to synthesize a texture sequence (or video) that satisfies the following properties:

1. *Appearance Similarity:* The synthesized sequence should be visually similar to the input texture. This is same as the appearance similarity component of our framework which was explicitly optimized in the previous chapter. This property, as also described earlier, aspires to maintain the structure of the input texture – shape, size, orientation, etc of its elements – in the synthesized texture sequence. In other words, we want the texture elements to flow as a unit as opposed to having individual pixels deform uninhibited by the flow. We use our texture energy metric that was described in the previous chapter and also in the section above to determine appearance similarity.

2. *Flow Consistency:* The texture sequence should be consistent with the input flow-field, *i.e.*, it should convey the motion represented by the flow-field. Conceptually, flow consistency will be maintained if the optic flow-field of the texture sequence matches the input flow-field. However, in practice we don't need to compute optical flow in the texture sequence. Instead, we measure flow consistency across two successive frames by warping the first frame using its flow-field and comparing it with the next.

These two properties define sub-goals for the synthesis process that may potentially be contradictory in nature. For example, if the input is an image texture and the first frame of the texture sequence has a high texture similarity score, then a sequence that exhibits no motion will easily satisfy the texture similarity property but it won't be consistent with the flow. On the other hand, a sequence generated by successively warping each frame of the texture sequence according to the flow-field will always be consistent with the flow, but it will cause the texture elements to distort unrecognizably or even disappear. Our formulation for controllable synthesis addresses this issue of competing objectives by having

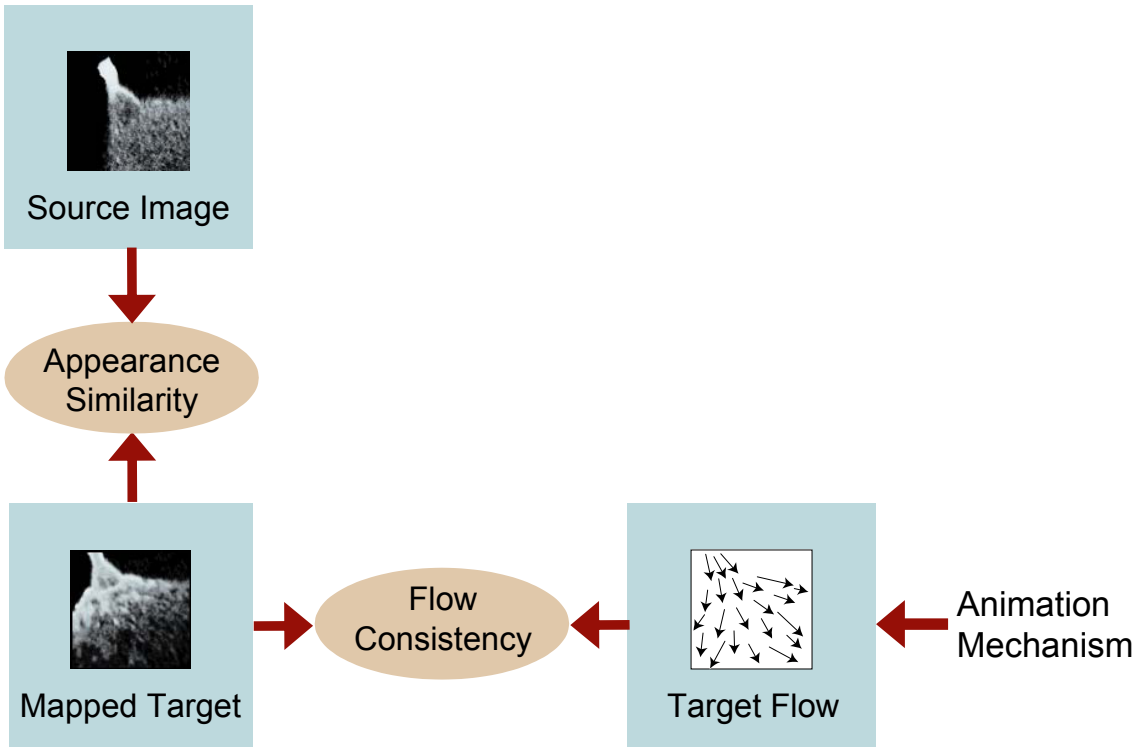


Figure 28: Specializing example-based rendering for synthesizing flowing image textures. Characteristic consistency is replaced by flow consistency. Source texture does not have its own flow, which simplifies the framework: no need to compute source flow, and flow consistency only depends on target flow and target appearance (mapped target).

a probability/energy term for each of this properties. The synthesized sequence is then a compromise between the two properties where the relative importance of one property over another can be controlled externally.

Before describing our technique for synthesizing flowing image textures, it is instructive to place it in the context of our example-based rendering framework. The specialization occurs in the characteristic consistency component of our framework, which is replaced by flow consistency. One major simplification that we achieve with the use of image textures as source imagery, is that flow consistency only needs to take into account the target flow and target appearance, *i.e.,* we can eliminate the source texture and source flow as inputs to flow consistency measurement. This is because the source texture does not have a flow of its own as it is static. This simplification is shown schematically in Figure 28.

6.2.1 Approach

To synthesize a sequence in which the texture moves according to a given input flow-field, we use a control term in our energy function that measures the consistency of the sequence with the flow – we call this term the flow energy. We synthesize the sequence frame-by-frame, so we need to define this term only for a single frame at a time. Let X now represent the frame being currently synthesized, and let X^- be the previous frame. A good way to measure flow consistency across the pair of frames X^- and X is to compute the optic flow between them and compare it with the input flow-field. However, it is difficult to incorporate optic flow computation within our framework because of its unreliability and non-linearity. We therefore use a simpler but equivalent measure based on the difference between X and the frame obtained by warping X^- using the input flow-field.

Let **f** denote the input flow-field using which we want to animate the texture. We allow for time-varying 2D flow-fields; hence, **f** is a sequence of 2D flow-fields $(f_1, f_2, \ldots\ f_{L-1})$, where L is the length of the texture sequence being synthesized, and f_i is the desirable flow-field between frame i and $i+1$. For a given pixel location p in frame i, $f_i(p)$ gives its

location in frame $i+1$ after it has been transported through the flow-field. We use f^- to denote the desirable flow-field between the previous and current frames, X^- and X. Let X^+ be the frame obtained by warping X^- using f^-. Then each pixel value in X^+ is determined by travelling backwards along f^- from that pixel and copying the resultant pixel value from X^-:

$$X^+(p) = X^-(q) \qquad q : f^-(q) = p$$

Once we have computed X^+, we can establish the flow energy as follows. Let $\mathbf{x}^+$ be the vectorized version of X^+, and $\mathbf{x}_p^+$ be its sub-vector corresponding to neighborhood $\mathcal{N}_p$. Then, the flow energy is defined as

$$U_c(\mathbf{x};\mathbf{x}^+) = (\mathbf{x}-\mathbf{x}^+)^T\mathbf{K}(\mathbf{x}-\mathbf{x}^+). \tag{27}$$

Here, $\mathbf{K}$ is a diagonal weighting matrix that allows us to assign a different weight to each pixel in the flow term. We have used it to assign weights to pixels based on intensity gradients. It is well known that motion perception and optic flow computation are more robust in the presence of high intensity gradients. Our weighting scheme exploits this fact by coercing the texture sequence to follow the input flow-field more faithfully in such high gradient regions. $\mathbf{K}$ can also be used to incorporate other heuristics into the energy function such as identifying *interesting* regions in the flow-field and weighting them more importantly relative to other regions.

The texture sequence is synthesized one frame at a time. To synthesize the current frame $\mathbf{x}$, we minimize its total energy:

$$U(\mathbf{x}) = U_t(\mathbf{x};\{\mathbf{z}_p\}) + \lambda U_c(\mathbf{x};\mathbf{x}^+).$$

The flow term, $U_c(\mathbf{x})$, attempts to keep the current frame close to the one obtained by warping the pervious frame, while the texture term, $U_t(\mathbf{x})$, tries to maintain its structural integrity by keeping it locally similar to the input texture. The two terms compete against each other by pulling each neighborhood sub-vector $\mathbf{x}_p$ towards potentially different sub-vectors $\mathbf{x}_p^+$ and $\mathbf{z}_p$. Note that $\mathbf{x}_p^+$ remains constant through successive iterations of the

minimization procedure even as $\mathbf{z}_p$ may change.

An observation about performing the optimization for this energy function is that the $U_c(\mathbf{x};\mathbf{x}^+)$ in (27) has a very similar form to the one defined in (23). Hence we can treat the synthesis of each frame as a soft-constrained synthesis problem. We synthesize the first frame of the sequence using regular texture synthesis. For each subsequent frame, the desirable pixel values at each location are obtained by warping the previous frame using its flow-field. The warped previous frame also acts as an initialization for the current frame. We then synthesize the frame using our controllable optimization approach for soft-constraints as discussed in the previous section. Algorithm 3 describes the flow-guided synthesis algorithm using Algorithm 1 and Algorithm 2 as sub-routines.

Algorithm 3 Flowing-guided Synthesis

$\mathbf{x}_1 \leftarrow$ texture synthesized using Algorithm 1
for frame $i = 2 : L$ **do**
  $\mathbf{x}^+ \leftarrow \text{warp}(\mathbf{x}_{i-1}, f_{i-1})$
  $\mathbf{x}_i \leftarrow$ output of Algorithm 2 using $\mathbf{x}^+$ as initialization and $U_c(\mathbf{x};\mathbf{x}^+)$ as control energy
end for

6.3 Handling Obstacles

We can easily adapt our technique to also handle obstacles that may be placed in the path of flow. Our main contribution here is to extend the texture optimization approach to perform synthesis in the presence of a mask. The mask specifies regions in the synthesized image which should *not* contain any texture. For example, an obstacle can be modeled as a mask in image space. We can also gradually fill the synthesized frame with texture over the course of the sequence by using a mask that changes over time. In this case, the mask represents not only the obstacle but also the empty region of the texture in each frame. Our contribution does not include the creation of flow-fields that would result in the presence of obstacles. We have used an interactive technique [69] to design flows that go around obstacles.

There are two modifications that we need to make to our technique in order to handle masks. These apply to the M-step and E-step respectively. We modify the M-step to allow searching for closest source neighborhoods by only matching a partial set of pixels. For each $\mathbf{x}_p$, the search now considers only those pixels that are not part of the mask. Recall that we originally performed this search using a hierarchical tree structure. Starting at the root of the tree, we traversed a path to a leaf node such that the nodes along the path matched best with $\mathbf{x}_p$. In the presence of a mask, we match $\mathbf{x}_p$ with tree nodes by only comparing unmasked pixels. Ideally, we would want to construct a tree for each possible set of partial pixels that would be used. However, that is impractical as there might be many such sets corresponding to different pixel neighborhoods $\mathcal{N}_p$. Empirically, we have found our approximation to yield satisfactory results.

We also modify the E-step where $\mathbf{x}$ is estimated by minimizing $U(\mathbf{x})$. We now consider only those pixels of $\mathbf{x}$ that are not part of the mask. The new energy function that we minimize is of the form

$$U(\mathbf{x}) = [\sum_{p \in X^\dagger} (\mathbf{x}_p - \mathbf{z}_p)^T \mathbf{K}_p^* (\mathbf{x}_p - \mathbf{z}_p)] + (\mathbf{x} - \mathbf{x}^+)^T \mathbf{K}^* (\mathbf{x} - \mathbf{x}^+). \quad (28)$$

The first term in this equation is the texture energy and the second term is flow energy. The mask is incorporated through $\mathbf{K}^*$ and $\mathbf{K}_p^*$. $\mathbf{K}^*$ is a binary ('0' or '1') diagonal matrix whose diagonal elements contain the complement (*logical not*) of the mask corresponding to each pixel location. Similarly, $\mathbf{K}_p^*$ is a (smaller) diagonal matrix that contains the mask-complement for only pixels in $\mathcal{N}_p$. These weighting matrices effectively *select* pixels from $\mathbf{x}$ that are not in the mask. The solution for $\mathbf{x}$ then modifies just the unmasked pixels. This procedure is repeated for every frame of the texture sequence, with potentially different masks.

Note that we can also use a gradient-based function for texture energy (as described in Section 5.3). In this case, the first term in (28) becomes $\sum_{p \in X^\dagger} (\mathbf{x}_p - \mathbf{z}_p)^T \mathbf{D}^T \mathbf{K}_p^* \mathbf{D} (\mathbf{x}_p - \mathbf{z}_p)$. The differentiation operator $\mathbf{D}$ outputs a vector containing the gradient at each edge in $\mathcal{N}_p$. Hence, the matrix $\mathbf{K}_p^*$ now encodes (binary) weights for each *edge* in $\mathcal{N}_p$ as opposed to

each pixel. Note that $\mathbf{K}_p^*$ contains a '0' entry for each edge whose either bounding pixel belongs to the mask. This ensures that the texture energy is not affected by any of the masked pixels.

6.4 Results

We have synthesized flowing image textures for a variety of flow-fields and texture examples. We have used our technique to synthesize texture animations for both time-varying and stationary flow-fields. Figure 29 shows some of the flow-fields that we have used to generate our results. It also shows the variability in the distortion that an image texture undergoes under the effect of different flows. Each frame was synthesized only at a single resolution and execution time per frame was between 20-60 seconds.

Figure 30 and Figure 31 show key-frames from texture sequences that were synthesized using a sink and a stream flow-field respectively. The structural elements of the texture are generally preserved even as it follows the flow-field. Notice that the shape of the marbles in the middle row of Figure 31 is maintained across the sequence. Some degeneracies do occur, as for example, in the center of the sink in Figure 30. The keys of the keyboard get smaller and smaller towards the sink. However, such a compromise may be necessary when the flow-field has inherent degeneracies as in the case of a sink. Also, the frames of the texture sequence are, at times, blurrier than the original texture, *e.g.*, in the middle row of Figure 31 (marbles texture). This is because our optimization approach leads to a blending of pixel values in order to simultaneously satisfy the texture and flow criteria. Also, note that for the nuts texture (Figure 31 bottom row), the three odd nuts in the first frame – that appear different from the rest of the texture – gradually disappear in the synthesized sequence. This happens because there was only one odd nut in the original texture. Therefore the algorithm has a hard time finding neighborhoods that look like that odd nut, and at the same time, align well with the motion induced by the flow-field.

In Figure 32, we compare the result of our approach to that obtained by applying a

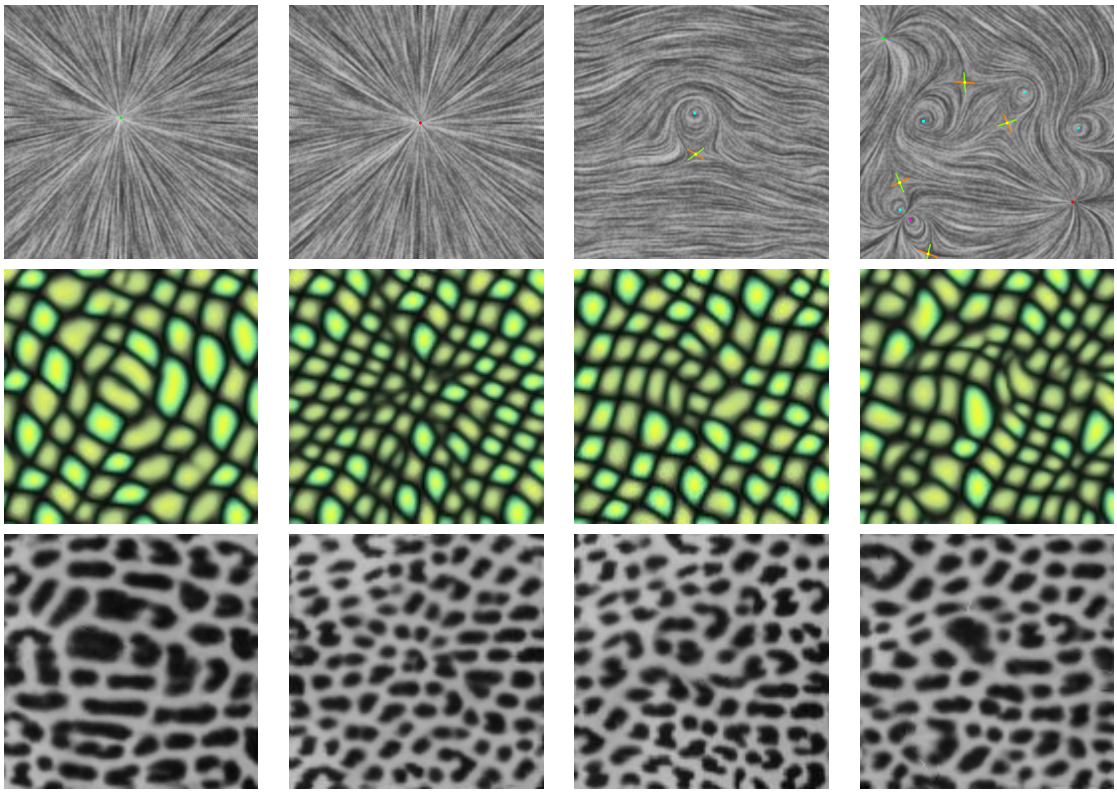


Figure 29: Flow-guided synthesis using different flow-fields. Shown is the 25th frame of each sequence (for both textures). All sequences for a given texture start with the same frame (not shown).

simple warp. Warping leaves holes in the texture and does not maintain similarity to the input sample. On the other hand, our technique generates texture sequences that convey the motion of the flow-field, while maintaining structural integrity of the texture at the same time. In the case of the keyboard example, the shape as well as orientation of the keys is maintained even as the entire texture rotates.

Figure 33 shows an example result for synthesizing texture sequences in the presence of obstacles. In this sequence, texture flows from left to right moving around an obstacle (shown in red). The blue region represents the empty portion of the texture in each frame. The texture used in the top row is a green scales texture, while the one used in the middle row is a snapshot of a flowing river. The frames of the texture sequence have been synthesized successfully in the presence of a mask. However, we observed that, in

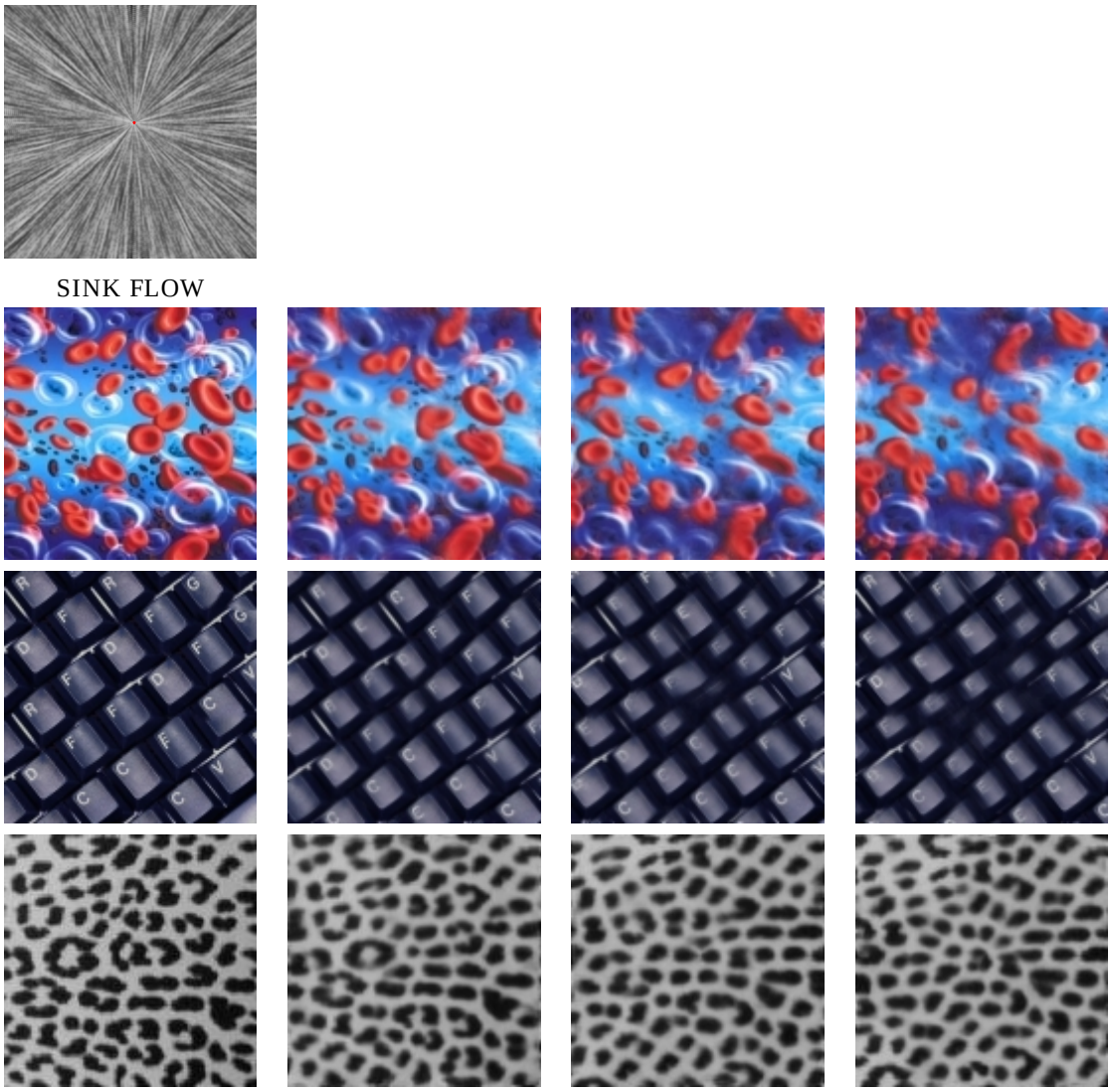


Figure 30: Animating texture using a flow-field. Shown are keyframes from texture sequences that follow a sink flow-field.

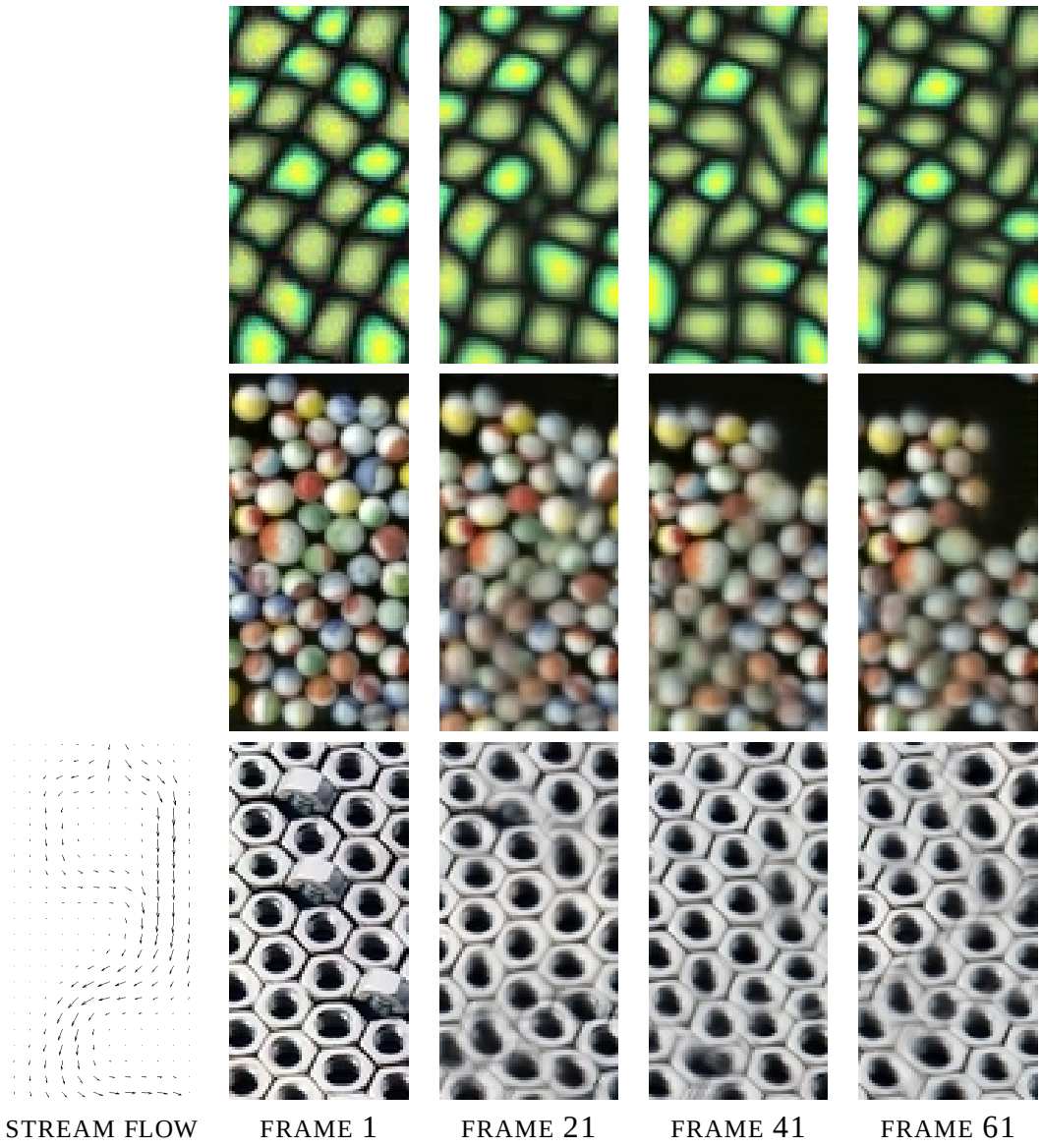


Figure 31: Animating texture using a flow-field. Shown are keyframes from texture sequences that follow a stream flow-field.

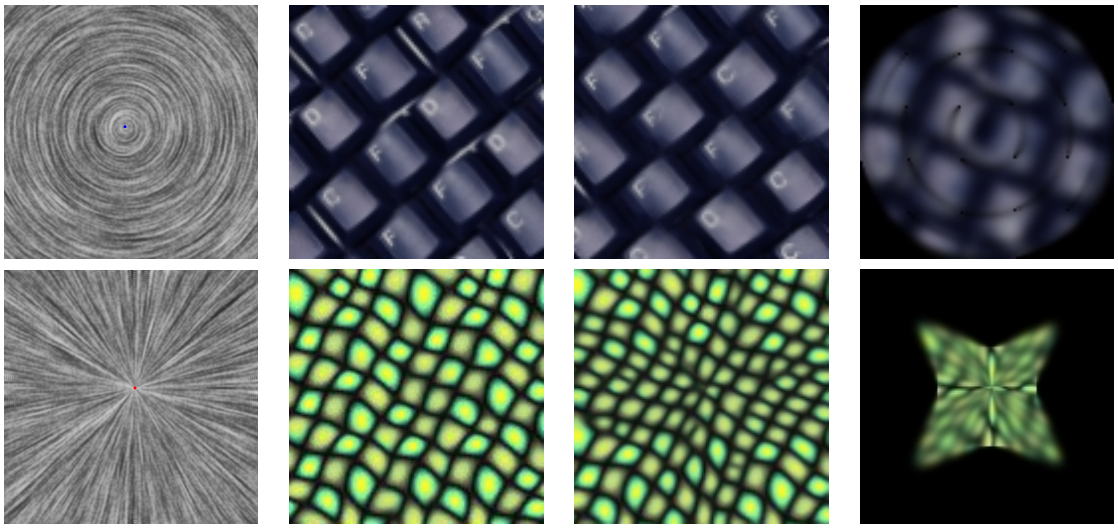


Figure 32: Comparison of our flow-guided synthesis results with simple warping. The keys on the keyboard maintain their orientation while rotating.

some regions below the obstacle, texture elements appear to go under the obstacle instead of going around it. We believe it to be more a consequence of an inaccurate flow-field – it was designed manually using an interactive program – than a limitation of the synthesis algorithm itself.

6.4.1 Discussion

Our flow-guided texture synthesis technique works on a variety of textures. However, the exact nature of the underlying mechanism is not obvious from the results. To better understand the process, we recall the texture optimization technique introduced in the last chapter, as it lies at the heart of our flow synthesis method. There are two aspects of the optimization that affect the outcome of flow-guided synthesis. *Firstly*, different texture elements (in the form of neighborhoods) are blended together to construct the output. Hence, when the texture follows a flow-field, the transition of texture elements from one frame to another occurs via blending of warped neighborhoods with each other. *Secondly*, the

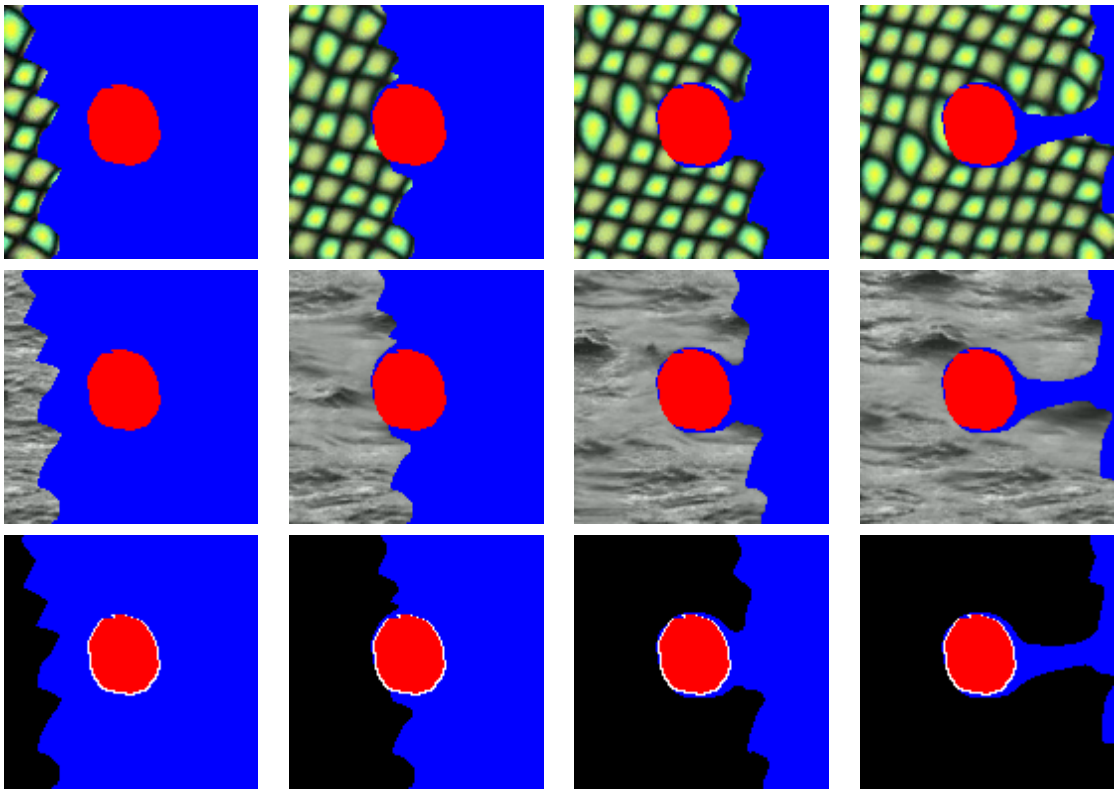
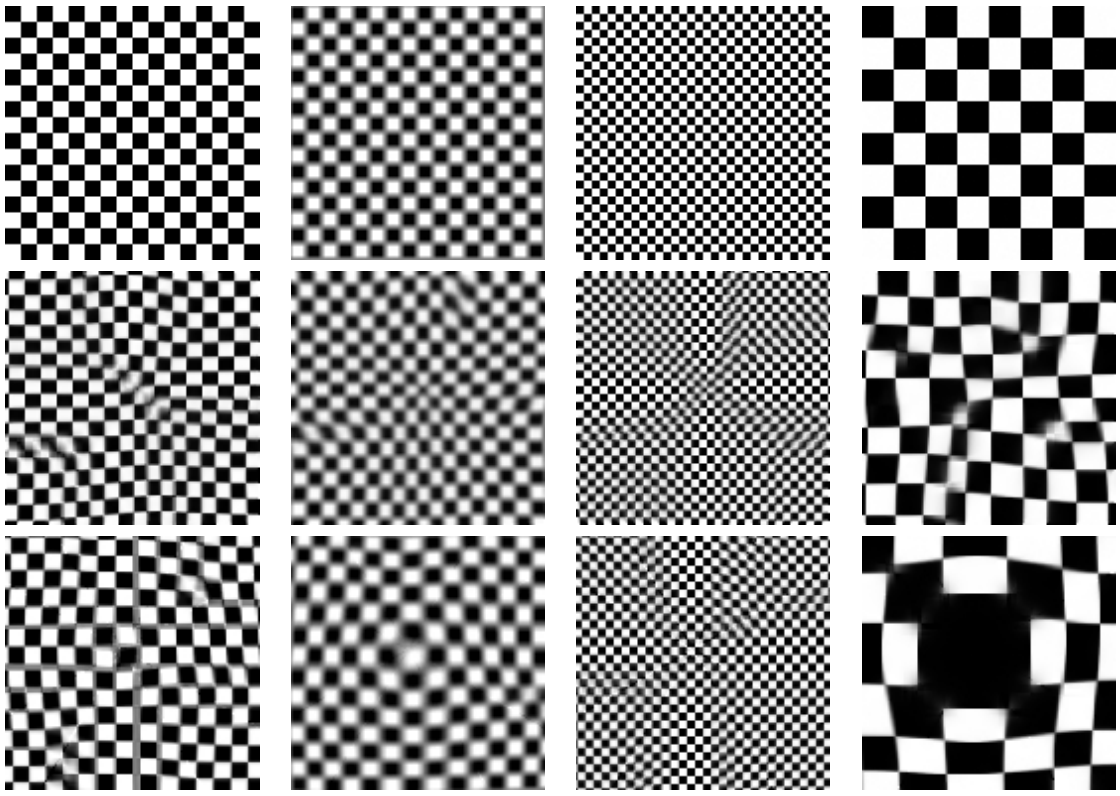


Figure 33: Handling obstacles. Shown (in the top two rows) are intermediate frames of sequences in which texture flows around an obstacle. The bottom row shows the mask (with its components) for each frame. The obstacle is shown in red with a white border around it (for clarity). The mask component representing empty space is shown in blue.



(a) ORIGINAL (b) FILTERED (c) HALF SCALE (d) DOUBLE SCALE

Figure 34: Flow-guided synthesis using a checkerboard pattern. The top row consists of the first frames for sequences corresponding to each input texture type – original, filtered, half-scale, or double-scale. The middle row shows an intermediate frame from the sequence synthesized with a sink flow-field, for each input texture type. The bottom row shows intermediate frames obtained with a source flow-field.

different neighborhood sizes used during synthesis affect the outcome of the algorithm. Therefore, the scale of the texture plays an important role in determining the quality of the output.

We conducted the following experiments to analyze our technique: using a checkerboard pattern as texture, we performed flow-guided synthesis with sink (convergent) and source (divergent) flows. Four variants of the original checkerboard pattern were used as texture: the original, a filtered (blurred) version of the original, the original at half-scale, and the original at double-scale. Figure 34 shows these variants and also the outcome of synthesis (intermediate frames) for the sink and source flow-fields.

A significant aspect of our technique is that texture elements are blended with each other to make a smooth transition from one frame to another. In the case of a checkerboard pattern, where the variation of color within an element – black to white and vice-versa – is stark, this blending can lead to noticeable blurring in the transition zone (Figure 34(a)). For the filtered version, the intermediate frames look more uniform (Figure 34(b)). This is because filtering leads to creation of smooth boundaries between white and black squares. The synthesis algorithm is then able to make use of these boundaries as transition zones when going from one frame to the next. Consequently, our technique is better suited for textures that allow (temporal) transition zones to be hidden between regions of constant or smoothly varying color.

The scale of the texture also plays an important role in determining quality of the synthesized sequence. The scale of the texture (size of texture elements) should be comparable to the largest neighborhood size used during synthesis. Also, it should be consistent with the scale of the flow-field, *i.e.,* the flow-field should not be too smooth or too rough with respect to the texture element size. The half and double scale texture results demonstrate this property. In the case of half-scale (Figure 34(c)), the texture elements are small, which means that large portions of these elements may appear to be part of a transition zone. This explains the large amount of gray in these results. Also, the neighborhood size used by

the synthesis algorithm is large relative to the smaller sized texture elements. This forces groups of texture elements (multiple squares in the checkerboard case) to maintain their shape across frames. Consequently, the distortion shown by the texture elements in order to accommodate flow is smaller, and is instead reflected in the form of more apparent transition zones. For the double scale case (Figure 34(d)), the largest synthesis neighborhood size is smaller than the texture element size. Hence, one can see that squares and rectangles, with sizes and shapes non-existent in the input, have been created in the output.

6.5 Summary

In this chapter, we presented a technique for controllable texture synthesis by extending the texture optimization approach presented in the previous chapter to incorporate additional control energy terms. We also derived this energy minimization formulation from the probabilistic principles that drive our example-based rendering framework. In particular, we developed a technique for synthesizing flowing image textures, *i.e.,* texture sequences that convey the motion represented by a flow-field but maintain the appearance of the input texture by preserving the structural integrity – shape, size, orientation, etc – of its constituent elements. This technique is a specific instantiation of our example-based rendering framework, in which the source imagery is an image texture and the characteristic being controlled is motion represented as flow. Using image textures as sources allows us to simplify the formulation by obviating the need for computing flow for the source texture, and also by allowing us to consider only target appearance and target flow in the design of similarity/energy metric for flow consistency. In the next chapter, we consider the case of video textures as source imagery with the target still controlled using flow. This requires using the complete formulation of example-based rendering to address the problem.

CHAPTER VII

FLOWING VIDEO TEXTURES

This chapter is devoted to extending the flow-guided texture animation technique presented in the last chapter to handle video textures – as opposed to just image textures – as source imagery. Using video as source texture also modifies the objectives that we need to satisfy during synthesis of flowing texture sequences. Recall that for flowing image textures, we stated two objectives: appearance similarity and flow consistency. Of these, the appearance similarity objective required the structural properties of the synthesized texture elements to stay close to that of the input texture. In the case of a video texture, the structural elements have a dynamic nature. They evolve over time, changing their appearance in terms of color, shape, size, orientation, etc. We want the texture elements in the synthesized sequence to also exhibit a dynamic evolution that is similar to the evolution in the source video. A related issue is that the source video has a flow of its own, *i.e.*, its elements may be moving spatially in addition to evolving over time. The synthesis technique needs to take this source flow into account.

Positioning this problem into the example-based rendering framework presented in Chapter 4, what we are trying to solve here is an instance of video-based rendering: using information from a source video, we want to render the appearance of a dynamic target scene – with motion (or flow) representing the dynamics of the scene. This is a more general instantiation of our framework than that with image textures, because we *need* to consider flow at the source – we could ignore it in the case of image textures. This is shown schematically in Figure 35. It shows a step that does flow extraction from source video. This source flow as well as the source video then need to be incorporated into the flow consistency measure.

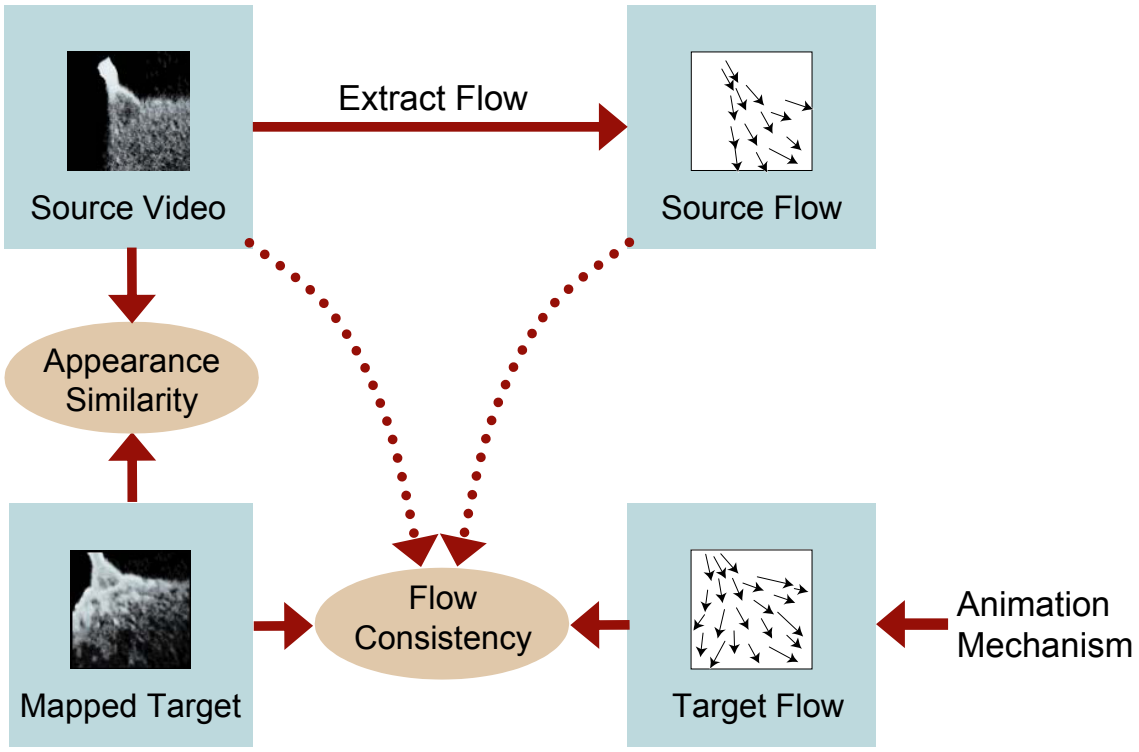


Figure 35: Specializing example-based rendering for synthesizing flowing video textures. Flow information from the source video needs to be extracted. Also, flow consistency now depends on the source appearance (video) and source flow in addition to target flow and target appearance (mapped target).

Ideally, the purpose of source flow extraction would be to decouple texture motion and texture evolution in the source video before performing synthesis. However, this turns out to be a difficult problem in the case of chaotically evolving video textures and requires extensive research of its own. Hence, we do not address the decoupling problem in this thesis. Instead, we postulate that the spatio-temporal appearance of texture elements in a video is governed by the motion of those elements. For synthesis, this assumption translates into the following observation: if we can find a spatio-temporal region in the source video that has flow similar to another spatio-temporal region in the target video, then the appearance of that source region can be copied over to the target region. This allows for two simplifications:

Firstly, we do not need to explicitly extract flow. As we will demonstrate, consistency of a source region with a given target flow region can be measured without explicitly computing source flow. However, it must be realized that this simplification is also a limitation of the approach. It means that we can only deal with target flows that are, at least locally, not much different from the flow in the source video. If the flows in the two videos are widely disparate, it will be difficult to convey the target flow in the synthesized sequence.

Secondly, the evolution of texture elements in the source can be replicated at the target simply by matching spatio-temporal regions of the source and the target. This is because we already enforce the flow in the two regions to be similar, and additionally, we have assumed that evolution of texture is governed by flow. Therefore, to achieve similar texture evolution as the source region, we simply need to copy the entire spatio-temporal region from the source to the target. The difficulty here lies in the fact that different overlapping spatio-temporal regions in the target may not agree with each other in the area of overlap. This would happen if we try to synthesize them by simply copying two different source regions that may be inconsistent with each other. Therefore, the meat of the problem lies in searching for source regions that have consistent appearance in the regions of overlap that result from copying them onto different target regions. At the same time, the flow in these

source regions should also match the flow in the target region where they are being copied.

7.1 Approach

We have extended the optimization-based synthesis approach that was presented in the last two chapters to also handle flow-guided synthesis using video textures. The basic difference in this modified approach and the previous one is that we now consider the entire 3D video volume as a single spatio-temporal texture, *i.e.*, synthesis is *not* performed frame-by-frame. Instead we define a flow (control) energy term that measures the consistency of the entire target video with the given target flow-field. Also, for the texture energy term, we consider spatio-temporal neighborhoods as was also done for unconstrained synthesis of video textures in Chapter 5. These spatio-temporal neighborhoods are same as the regions mentioned above. For each target neighborhood under consideration, we search for source neighborhoods that match the flow in that target neighborhood. Since we use overlapping neighborhoods in our texture energy formulation, the picked source neighborhood is the result of a compromise between matching the flow and matching the appearance of overlapping neighborhoods. Also, the appearance assigned to the target pixels is an interpolation between pixels values of overlapping neighborhoods – recall that this was also the case when we performed texture synthesis using optimization or synthesized flowing image textures. This interpolation is such that it minimizes the total energy of the video for a given set of source neighborhoods. The total energy takes into account both the evolution of texture elements (through texture energy) as well as its consistency with the given target flow (through flow energy).

Using the notation from the previous chapter, let **f** denote the given target flow field, defined as a sequence of 2D flow fields $(f_1, f_2, \ldots\ f_{L-1})$. Recall that for a given pixel location p in frame i, $f_i(p)$ gives its location in frame $i+1$ after it has been transported through the flow field. Since we want to synthesize the entire spatio-temporal video volume at once, we use X to denote the complete texture sequence – as opposed to denoting a

single frame in the sequence. We use X_i and X_{i+1} to denote two consecutive frames of the sequence. Note that X is now a 3D volume and X_i represents a 2D slice within this volume. We denote pixels in X as (p,i) where p is the pixel location in 2D and i is the frame number. To define the flow energy, we observe that we want the pixels in X_i and X_{i+1} that are related via f_i, to be similar in appearance. Pixels (p,i) and $(q,i+1)$ are said to be related via f_i if $q = f_i(p)$. Intuitively, this means that we want pixels to retain their appearance as much as possible while moving along the flow field. Hence, we define the flow energy function as

$$U_c(\mathbf{x};\mathbf{f}) = \sum_{i \in 1:L-1} \sum_{(p,i) \in X_i} (\mathbf{x}(p,i) - \mathbf{x}(q,i+1))^2, \;\; q = f_i(p). \tag{29}$$

In principle, this equation is the same as that used for flowing image textures (see (27)). However, it is defined over the entire 3D volume $\mathbf{x}$. Therefore, all pixels are treated as variables which need to be determined at the same time. On the contrary, in (27), $\mathbf{x}$ refers to a single frame of the sequence and its energy is defined with respect to the warped previous frame $\mathbf{x}^+$ which remains fixed throughout the optimization procedure. One can visualize the flow energy function by thinking of pixels as masses that are connected through springs. The springs are attached only between those masses (pixels) that are connected via the flow-field. We need to compute the stable position (color value) of these masses (pixels). In (27), the position of some of the masses (pixels from previous frame) is kept fixed, while the rest (pixels in current frame) are free to move. Note that there are no springs between masses that are mobile. However, in (29), all the masses are free to move. They will be pulled in different directions as determined by the texture energy, subsequently resulting in a spring force that will try to keep the connected masses as close as possible. We want to find a stable state in which the total spring-mass energy is minimized. This total energy that we wish to minimize is

$$U(\mathbf{x}) = U_t(\mathbf{x};\{\mathbf{z}_p\}) + \lambda U_c(\mathbf{x};\mathbf{f}), \tag{30}$$

where $U_t(\mathbf{x};\{\mathbf{z}_p\})$ is the same as in (18). We have used the same iterative algorithm for optimizing this energy function that was used in the last chapter for flowing image textures.

However, the E and M steps of the algorithm now have different interpretations.

Recall that in the M-step, we find the closest source neighborhood sub-vector $\mathbf{z}_p$ corresponding to each target neighborhood $\mathbf{x}_p \in X^\dagger$. In the video case, these are 3D neighborhoods. We discussed a modification of the M-step in the previous chapter that allows us to search for source neighborhoods that may already be consistent with the control criteria. In this case, this corresponds to searching for source neighborhoods $\mathbf{z}_p$ that have the same perceived flow as the target neighborhood $\mathbf{x}_p$. For each $\mathbf{x}_p$, we find $\mathbf{z}_p$ that minimizes the following:

$$U_p(\mathbf{x};\mathbf{z}_p) = \|\mathbf{x}_p - \mathbf{z}_p\|^2 + \lambda U_c(\mathbf{y};\mathbf{f}), \tag{31}$$

where $\mathbf{y}$ is obtained by replacing the pixels of $\mathbf{x}$ in neighborhood $\mathcal{N}_p$ with $\mathbf{z}_p$. Note that we are *searching* for the sub-vector $\mathbf{z}_p$ that minimizes (31), *i.e.*, $\mathbf{z}_p$ is the variable while $\mathbf{x}$ stays fixed. This means that $\mathbf{y}$ changes across the different candidates for $\mathbf{z}_p$ only at the pixels in the neighborhood $\mathcal{N}_p$. Expanding the term $U_c(\mathbf{y};\mathbf{f})$ in (31) using (29), we get

$$U_c(\mathbf{y};\mathbf{f}) = \sum_{i \in 1:L-1} \sum_{(r,i) \in X_i} (\mathbf{y}(r,i) - \mathbf{y}(q,i+1))^2, \;\; q = f_i(r).$$

Here, r is used as a pixel index in place of p to avoid confusion with the index of the target neighborhood $\mathcal{N}_p$ under consideration. The energy represented by this equation measures the consistency of $\mathbf{y}$ with respect to the flow-field $\mathbf{f}$. Also, as mentioned above, the pixels of $\mathbf{y}$ that correspond to neighborhood $\mathcal{N}_p$ are same as the pixels of the candidate source neighborhood $\mathbf{z}_p$. This implies that this energy term implicitly measures the consistency of $\mathbf{z}_p$ with respect to $\mathbf{f}$ at the neighborhood $\mathcal{N}_p$. Thus, in the M-step, we are searching for source neighborhoods that are consistent with the flow at the corresponding target neighborhood in addition to being similar to the target neighborhood in appearance.

In the E-step, we minimize (30) w.r.t. the entire video volume $\mathbf{x}$. This again leads to a sparse linear system of equations. The sparsity structure of the system matrix is governed by the flow-field $\mathbf{f}$. Each row and each column of the system matrix corresponds to a pixel in the video volume. Each row of the matrix contains a coefficient in the column

corresponding to the pixel to which the row's pixel is connected via the flow-field. We solve this linear system using pre-conditioned conjugate gradients. Intuitively, this minimization procedure interpolates the source neighborhoods $\mathbf{z}_p$ such that the pixels connected via $\mathbf{f}$ are as similar to each other as possible; this is done while also maintaining the spatio-temporal appearance of the source neighborhoods in the synthesized target.

Another algorithmic detail of relevance here is that we modify the initialization step in which an estimate of the synthesized video is first created. In the case of unconstrained video synthesis, we had used random source neighborhoods to form the set $\{\mathbf{z}_p\}$. However, in the case of flowing video textures, we already know the desirable flow for each target neighborhood, but not its appearance. Assigning random source neighborhoods to the target is not desirable since the flow in that neighborhood may not match the flow in the target. Hence, we search for source neighborhoods that match the flow in the corresponding target neighborhood without regard to appearance, *i.e.*, we only use the second term, $U_c(\mathbf{y};\mathbf{f})$, of (31) in the search for closest source neighborhoods $\mathbf{z}_p$ during initialization.

7.2 Results and Discussion

We have experimented with various flow-fields and source video textures for synthesizing flowing video textures. Some examples are shown in Figure 36. As expected, videos for which the desirable flow is available in the source generally give better results than those for which the source and target flow do not match. If the flows are very different, the synthesized texture sequence is not able to convey the motion of the given flow-field. One solution that we have tried to handle this is to give a higher weight λ to the flow energy term. However, this leads to excessive blurring in the synthesized video due to interpolation between spatio-temporal neighborhoods that may not match well with each other.

The best results that we have obtained are for the pond video, since the source flow in this case consists of standing waves that cause image motion in all directions. However, even our best result for flowing video textures – rotating pond – is not as convincing as

many of our results for flowing image textures. The primary reason for this is that it is very hard to find regions in the source that match the target flow as well as have spatio-temporal appearance that is consistent with neighboring regions in the target. This is related to the assumption that we made in the beginning of this chapter, that the appearance of a spatio-temporal neighborhood is governed by the flow in that region. For many complex videos, this assumption is not true; texture in different regions may be evolving differently even if the flow in those regions is similar.

7.2.1 Analysis

To better understand these limitations, we need to analyze the synthesis process more carefully. The optimization proceeds by picking 3D spatio-temporal neighborhoods from the source that match well with the corresponding target neighborhoods in terms of flow and appearance. While the target flow remains fixed throughout the optimization, the target appearance (which is the same as the synthesized video) as well as the picked source neighborhoods change at each iteration. The optimization converges to a solution when the source neighborhoods and target appearance stop changing. What we want to achieve through this optimization is to converge onto a set of source neighborhoods that match well with each other in terms of appearance but still have the same flow as the target. The converged solution might present a number of scenarios with respect to the configuration of these source neighborhoods. Consider two source neighborhoods that overlap with each other when placed in the target video. Remember that these are 3D spatio-temporal neighborhoods. Lets look at some of the possible scenarios:

The first scenario we consider is where the flow in the neighborhoods matches the target flow, and the neighborhoods are also consistent with each other in terms of appearance, *i.e.,* they have similar color values in the region of overlap. This is the ideal scenario and we would like it to hold everywhere in the synthesized video. However, if there is significant difference between the target and source flow, then it is difficult to achieve this

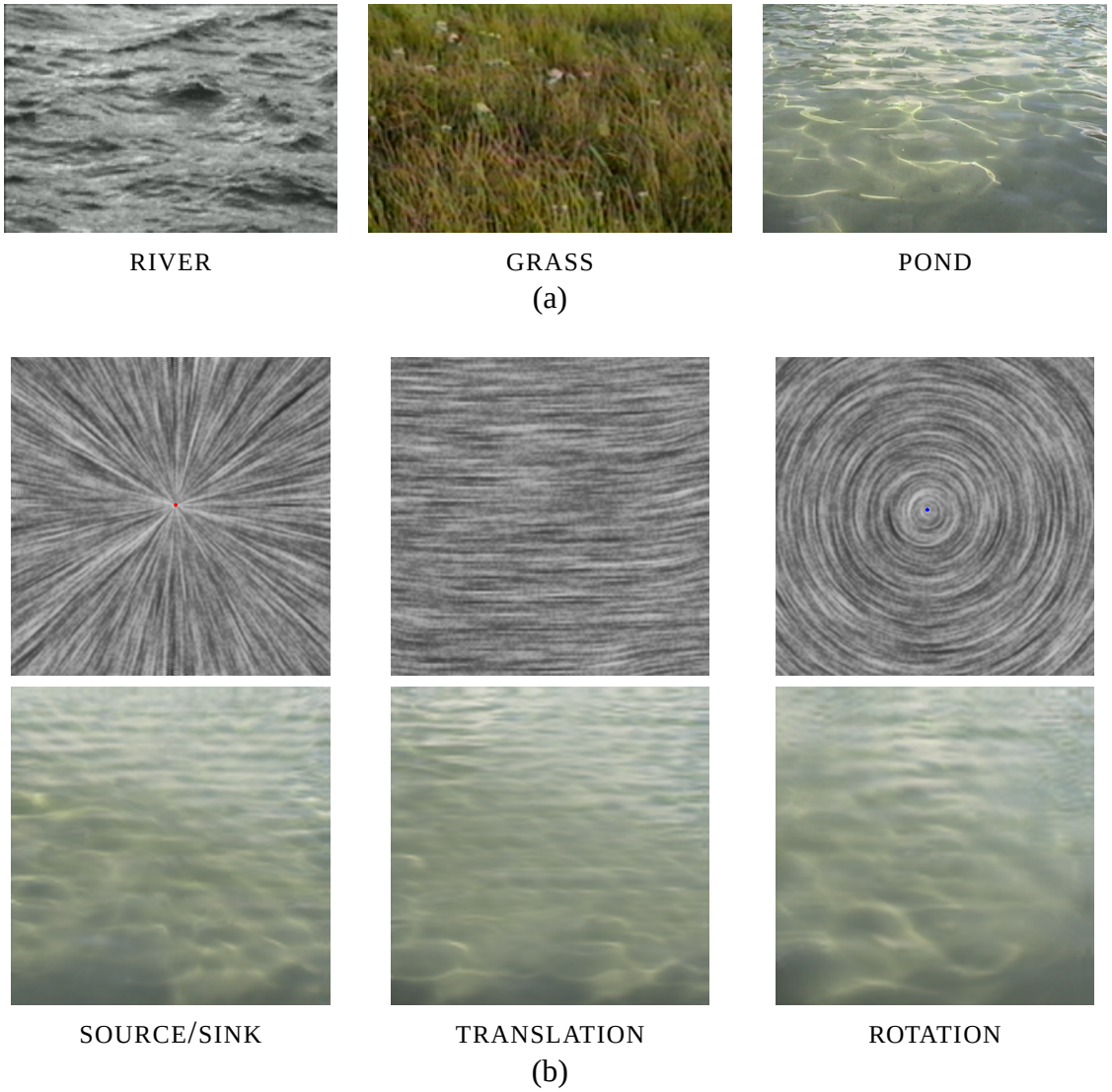


Figure 36: (a) Examples of video textures that were experimented with for synthesizing flowing video textures. (b) Flow fields that were used for synthesis are shown in the top row. The bottom row shows frames from synthesized pond sequences corresponding to each flow field.

property everywhere. What ends up happening is that some regions in the synthesized video exhibit this scenario while others do not. In the best case, such *good* regions encompass a large portion of the synthesized video. In the case of the translating pond result shown in Figure 36, the translation is most evident only in the top half of the frame.

The second scenario is where the source neighborhoods are consistent with each other in terms of appearance but the flow in those neighborhoods does not match the desired flow at the target. This is a point of failure: the composite appearance in the region where these neighborhoods lie will now exhibit a flow which will not be consistent with underlying target flow. In other words, while the texture evolution in these neighborhoods will be similar to the source, the objective of controlling the motion in the synthesized video with the given target flow will not be satisfied. This is what happens in the bottom half of the translating pond result. There is visible apparent motion, but it is not the translational motion that is desired.

The third scenario is where the flow in the source neighborhoods is consistent with the target flow, but the neighborhoods themselves are not consistent with each other in terms of appearance, *i.e.,* the two neighborhoods do not agree with each other in their region of overlap. The outcome of this mismatch between neighborhoods is that resulting video gets blurred in the region of overlap. Thus the texture appearance and evolution is not as similar to the source as desired. An indirect consequence of this blurring is that the flow in the overlap region may become less discernible even if it matches the target. The rotating pond result in Figure 36 demonstrates this scenario. The synthesized video generally follows the rotational field, however blurring in certain regions makes the motion subtle and hard to perceive.

The last scenario is the worst case where neither the flow in source neighborhoods matches target flow nor is the appearance of these neighborhoods consistent with each other. The regions in which this occurs are usually very blurry and/or may exhibit arbitrary motion that has nothing to do with the target flow.

7.3 Potential Improvements

From our discussion, it is evident that the approach we have used for synthesizing flowing video textures is heavily dependent on the agreement between input data (source flow and appearance) and output control variables (target flow). In this section, we discuss the potential improvements and related issues that need to be explored to address this limitation.

One solution to the problem of mismatch between source and target flow is to artificially increase the input data by considering transformations of the source video like rotation and scaling. If we make use of a sufficiently large number of rotations and scales of the source video, then the variety of flow available at the source suddenly increases multiple times. Of course, we may lose the ability to maintain the orientation and size of texture elements but that may be acceptable. The main issue with such an approach is its computational cost. We will have to search for source neighborhoods from an enormous amount of video data. Moreover, even if we are able to handle it computationally, this would work only for videos that do not violate our assumption that texture evolution is governed by the flow in the video.

In order to handle videos that may have little correlation between flow and texture evolution, we need to address the decoupling of these two processes. Flow is responsible for the change in texture appearance due to transport of texture elements along the flow. On the other hand, texture evolution is the change that these elements undergo over time irrespective of the flow. Decoupling these process is a challenging problem because without any constraints on either, any one process may be sufficient to explain the entire appearance change. A simple way of trying to solve decoupling is to first compute flow using an optic flow technique and then determine evolution as the residual appearance change remaining after taking flow into account. However, the issue here is that most optic flow techniques assume brightness constancy, *i.e.*, they assume that any change in pixel appearance is due to flow. This is obviously not the case for textural video. In fact, this assumption tries to nullify the residual appearance change that we are trying to estimate in the first place. To

tackle this, a reasonable strategy is to make prior assumptions about the flow and residual appearance/texture evolution. For example, we can favor smooth flow fields and compact descriptions of residual appearance.

Once such a decomposition is available, it can be used to design a likelihood (flow energy) function for measuring consistency with the target flow. An observation is that under this decomposition, one can think of appearance change from one frame to the next as a two step process. In the first step, the texture in the current frame gets warped according to the flow field. In the next step, the appearance of the texture evolves based on a process that governs residual appearance. The likelihood term can be designed such that it favors those target videos that can be best explained as a concatenation of these two operations applied to each frame of the video.

7.4 Summary

In this chapter, we discussed the problem of synthesizing motion controlled video textures. This involves synthesizing texture sequences that use appearance from a source video but whose motion is controlled by a flow field. We presented a technique for solving this problem that extends the optimization-based approach for synthesizing flowing image textures presented in the previous chapter. We assume that evolution of texture elements in the video is governed by its flow. This allows us to perform synthesis by copying spatio-temporal regions in the source video that match well with the desired target flow. However, we do this matching without explicitly computing the source flow. Instead, we search for matching source neighborhoods by computing their consistency with target flow on the fly. We also discussed the limitations of this approach that arise due to our assumptions and also because we ignore the decoupling between texture flow and texture evolution in the source video. We also proposed potential extensions that would alleviate most of the problems associated with our current approach.

CHAPTER VIII

CONCLUSIONS AND FUTURE WORK

This thesis explores synthesis by example as a paradigm for rendering real-world phenomena. In particular, we consider phenomena that can be visually described as texture. We exploit, for synthesis, the self-repeating nature of the visual elements constituting these texture exemplars. We present techniques for unconstrained as well as constrained/controllable synthesis of textures.

For unconstrained synthesis, we present two robust techniques that can perform spatio-temporal extension, editing, and merging of image as well as video textures. In one of these techniques, large patches of input texture are automatically aligned and seamless stitched with each other to generate realistic looking images and videos. The second technique is based on iterative optimization of an energy function that measures the quality of the synthesized texture with respect to the input exemplar.

We also present a technique for controllable texture synthesis. In particular, it allows for generation of motion-controlled texture animations that follow a specified flow field. Animations synthesized in this fashion maintain the structural properties like local shape, size, and orientation of the input texture even as they move according to the specified flow. We cast this problem into an optimization framework that tries to simultaneously satisfy the two (potentially competing) objectives of similarity to the input texture and consistency with the flow field. This optimization is a simple extension of the approach used for unconstrained texture synthesis.

A general framework for example-based synthesis and rendering is also presented. This framework provides a *design space* for constructing example-based rendering algorithms. The goal of such algorithms would be to use texture exemplars to render animations for

which certain behavioral characteristics need to be controlled. Our motion-controlled texture synthesis is an instantiation of this framework where the characteristic being controlled is motion represented as a flow field.

8.1 Future Directions

There are multiple directions for future research that can be explored from where this thesis ends. Some of these are immediate extensions of the techniques that we have presented, while others build on our general framework for example-based synthesis.

8.1.1 Decoupling Flow and Evolution in Video

In Chapter 7, we had discussed the limitations of our current approach for synthesizing flowing video textures and also suggested potential improvements. These included decoupling the appearance change in the source video into texture flow and texture evolution. We believe that solving this decoupling problem will go a long way in improving the quality of results for motion-controlled video synthesis. A related research direction is the design of likelihood (or energy) functions for measuring flow consistency that make use of this decoupling.

8.1.2 Other Characteristics besides Motion

We have primarily used flow for controlling the behavior of the synthesized texture animations. However, the probabilistic example-based rendering formulation in Chapter 4 was presented from the point-of-view of general characteristics. A future research direction is to make use of this formulation to devise algorithms for characteristics other than flow. We will need to make use of domain knowledge specific to the characteristic of interest in the design of likelihood (or control energy) functions, but the basic principles of the formulation would still apply. Examples of these other characteristics include the following:

Figure 37: Breaking Wave.

Shape: For motion-controlled synthesis, we ignored 3D scene information and considered motion only in the image plane. However, in many situations, it is also desirable to control the shape (in 3D) of the entity being rendered. If this entity is a fixed object, then the problem is not very interesting, as it can be solved trivially through texture mapping, *i.e.,* by defining a mapping between surface points and texture pixels. However, if the shape is changing dynamically over time, then we need to define this mapping for each intermediate shape. In the process, we also want to preserve the structural integrity of the texture as well as keep it temporally coherent. Consider the example of a breaking wave as shown in Figure 37. If we want to synthesize a new wave using example video of another wave, we need to establish a correspondence between the shape and appearance of waves. We can use surface normals to represent shape in the same way we used flow to represent motion.

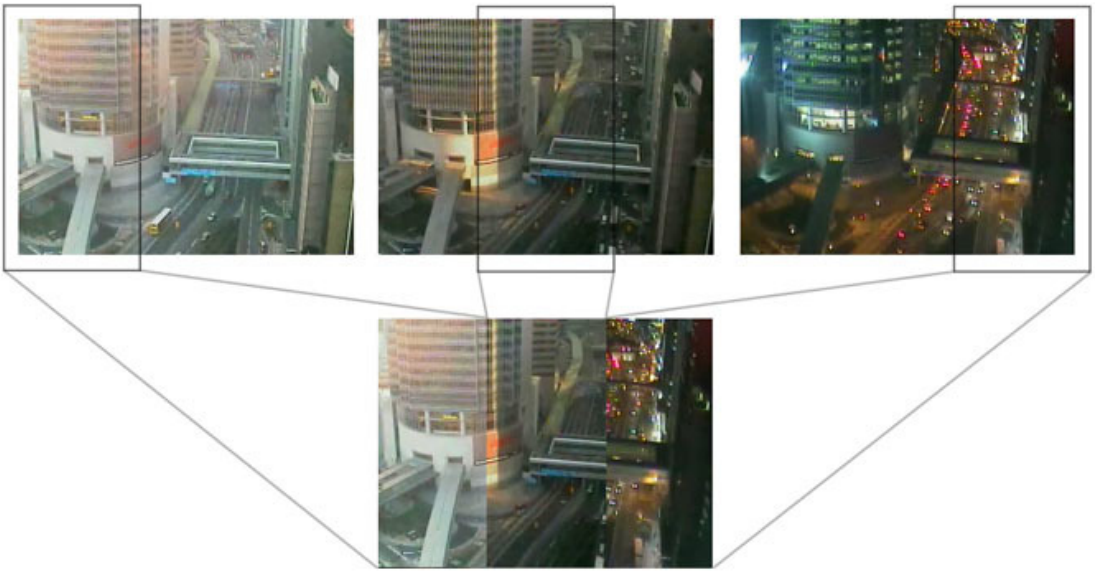


Figure 38: Changing time of the day (courtesy: Mike Terry).

The missing components then include a method for extracting these surface normals from video as well as a likelihood function that measures consistency of the synthesized video with the dynamic shape that we want it to depict.

Illumination, Temperature, Density: Another characteristic of interest is illumination. Lets say we have imagery of a scene taken at different times of the day as shown in Figure 38. The texture of entities like buildings, roads, cars, etc undergoes a systematic change over time as governed by the illumination in the scene. One might be interested in using such imagery to synthesize novel scenes that depict different times of the day, or better yet, scenes in which we have spatial control over illumination. We can also use other characteristics like temperature and density to control synthesis. These properties determine the local appearance of textures like fire and smoke (Figure 39). As an example, one might want to control the density of smoke or the incandescence of fire in a region. It would be interesting to explore an example-based rendering technique that works in conjunction with fire or smoke simulation methods that generate these characteristics at each point in

Figure 39: Fire and Smoke.

the scene.

8.1.3 Interactive Video Editing

We envision that example-based synthesis would provide a platform to develop tools for interactive video editing. Many natural scenes captured on video contain textural elements like water, fire, crowds, traffic, etc. We intend to develop methods to edit these videos interactively by extracting and manipulating their behavioral characteristics and then feeding them to an example-based rendering system. For example, we can extract trajectories of moving people from crowd videos, edit these trajectories and synthesize a new video in which people move along these modified trajectories. A more complex operation may be

needed if we want to edit a water video; lets say, by putting an obstacle in the path of a river stream. An interesting direction to explore here is to combine fluid simulation and example-based rendering, where fluid simulation generates realistic behavior for the motion of the water as it goes around the obstacle while example-based rendering imparts it a natural looking appearance.

REFERENCES

[1] ASHIKHMIN, M., "Synthesizing natural textures," *2001 ACM Symposium on Interactive 3D Graphics*, pp. 217–226, March 2001. ISBN 1-58113-292-1. 2.2, 2.4

[2] BAR-JOSEPH, Z., EL-YANIV, R., LISCHINSKI, D., and WERMAN, M., "Texture mixing and texture movie synthesis using statistical learning," *IEEE Transactions on Visualization and Computer Graphics*, vol. 7, no. 2, pp. 120–135, 2001. 2.2, 3.5, 3.5

[3] BHAT, K. S., SEITZ, S. M., HODGINS, J. K., and KHOSLA, P. K., "Flow-based video synthesis and editing," *ACM Transactions on Graphics (SIGGRAPH 2004)*, vol. 23, August 2004. 2.4

[4] BOYKOV, Y., VEKSLER, O., and ZABIH, R., "Fast approximate energy minimization via graph cuts," in *International Conference on Computer Vision*, pp. 377–384, 1999. 2.3, 3.1.1

[5] BRAND, M., "Subspace mappings for image sequences," in *Statistical Methods in Video Processing*, JUNE 2002. 2.2

[6] BREGLER, C., COVELL, M., and SLANEY, M., "Video rewrite: Driving visual speech with audio," *Proceedings of SIGGRAPH 97*, pp. 353–360, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California. 2.4

[7] BROOKS, S. and DODGSON, N. A., "Self-similarity based texture editing," *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2002)*, vol. 21, pp. 653–656, July 2002. 2.4, 3.4

[8] BURT, P. J. and ADELSON, E. H., "A multiresolution spline with application to image mosaics," *ACM Transactions on Graphics*, vol. 2, no. 4, pp. 217–236, 1983. 2.3, 3.3

[9] BURT, P. J. and ADELSON, E. H., "The laplacian pyramid as a compact image code," *IEEE Transactions on Communications*, vol. COM-31,4, pp. 532–540, 1983. 2.3

[10] CHEN, S. E., "Quicktime vr - an image-based approach to virtual environment navigation," *Proceedings of SIGGRAPH 95*, pp. 29–38, August 1995. ISBN 0-201-84776-0. Held in Los Angeles, California. 2.1

[11] COHEN, M. F., SHADE, J., HILLER, S., and DEUSSEN, O., "Wang tiles for image and texture generation," *ACM Transactions on Graphics, SIGGRAPH 2003*, vol. 22, no. 3, pp. 287–294, 2003. 2.2

[12] COLEMAN, D., HOLLAND, P., KADEN, N., KLEMA, V., and PETERS, S. C., "A system of subroutines for iteratively reweighted least squares computations," *ACM Trans. Math. Softw.*, vol. 6, no. 3, pp. 327–336, 1980. 5.2

[13] CROW, F. C., "Summed-area tables for texture mapping," in *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pp. 207–212, 1984. ISBN 0-89791-138-5. 3.3

[14] DEBONET, J. S., "Multiresolution sampling procedure for analysis and synthesis of texture images," *Proceedings of SIGGRAPH 97*, pp. 361–368, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California. 2.2, 2.3, 1

[15] DELLAERT, F., KWATRA, V., and OH, S. M., "Mixture trees for modeling and fast conditional sampling with applications in vision and graphics," in *IEEE Computer Vision and Pattern Recognition*, 2005. 2

[16] DORETTO, G. and SOATTO, S., "Editable dynamic textures," in *IEEE Computer Vision and Pattern Recognition*, pp. II: 137–142, 2003. 2.4

[17] EFROS, A. and LEUNG, T., "Texture synthesis by non-parametric sampling," in *International Conference on Computer Vision*, pp. 1033–1038, 1999. 2.2, 2.3, 1, 5.2

[18] EFROS, A. A. and FREEMAN, W. T., "Image quilting for texture synthesis and transfer," *Proceedings of SIGGRAPH 2001*, pp. 341–346, August 2001. ISBN 1-58113-292-1. (document), 2.2, 2.3, 2.4, 1, 3.1, 3.4, 10

[19] ELKAN, C., "Using the triangle inequality to accelerate k-means," in *International Conference on Machine Learning*, 2003. 2

[20] EZZAT, T., GEIGER, G., and POGGIO, T., "Trainable videorealistic speech animation," in *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pp. 388–398, ACM Press, 2002. 2.4

[21] FITZGIBBON, A., WEXLER, Y., and ZISSERMAN, A., "Image-based rendering using image-based priors," in *International Conference on Computer Vision*, 2003. 2.3

[22] FORD, L. and FULKERSON, D., *Flows in Networks*. Princeton University Press, 1962. 3.1

[23] FREEMAN, W. T., JONES, T. R., and PASZTOR, E. C., "Example-based super-resolution," *IEEE Comput. Graph. Appl.*, vol. 22, no. 2, pp. 56–65, 2002. 2.3

[24] GORTLER, S. J., GRZESZCZUK, R., SZELISKI, R., and COHEN, M. F., "The lumigraph," *Proceedings of SIGGRAPH 96*, pp. 43–54, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana. 2.1

[25] GREIG, D., PORTEOUS, B., and SEHEULT, A., "Exact maximum a posteriori estimation for binary images," *Journal of the Royal Statistical Society*, vol. Series B, 51, pp. 271–279, 1989. 2.3

[26] GUO, B., SHUM, H., and XU, Y.-Q., "Chaos mosaic: Fast and memory efficient texture synthesis," Tech. Rep. MSR-TR-2000-32, Microsoft Research, 2000. 2.2

[27] HAMMERSLEY, J. M. and CLIFFORD, P., "Markov field on finite graphs and lattices." 1971. 5.1

[28] HEEGER, D. J. and BERGEN, J. R., "Pyramid-based texture analysis/synthesis," *Proceedings of SIGGRAPH 95*, pp. 229–238, August 1995. ISBN 0-201-84776-0. Held in Los Angeles, California. 2.2

[29] HERTZMANN, A., JACOBS, C. E., OLIVER, N., CURLESS, B., and SALESIN, D. H., "Image analogies," *Proceedings of SIGGRAPH 2001*, pp. 327–340, August 2001. ISBN 1-58113-292-1. 2.4

[30] HOWE, D., *Free Online Dictionary of Computing*. http://www.foldoc.org/. 1

[31] JOHNSON, S. C., "Hierarchical clustering schemes," *Psychometrika*, vol. 2, pp. 241–254, 1967. 2

[32] JOJIC, N., FREY, B., and KANNAN, A., "Epitomic analysis of appearance and shape," in *International Conference on Computer Vision*, 2003. 2.3

[33] KILTHAU, S.L., DREW, M., and MOLLER, T., "Full search content independent block matching based on the fast fourier transform," in *ICIP02*, pp. I: 669–672, 2002. 3.3

[34] KWATRA, V., ESSA, I., BOBICK, A., and KWATRA, N., "Texture optimization for example-based synthesis," *ACM Transactions on Graphics, SIGGRAPH 2005*, August 2005. 5

[35] KWATRA, V., SCHÖDL, A., ESSA, I., TURK, G., and BOBICK, A., "Graphcut textures: Image and video synthesis using graph cuts," *ACM Transactions on Graphics, SIGGRAPH 2003*, vol. 22, pp. 277–286, July 2003. 1

[36] LEVOY, M. and HANRAHAN, P., "Light field rendering," *Proceedings of SIGGRAPH 96*, pp. 31–42, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana. 2.1

[37] LI, S. Z., *Markov Random Field Modeling in Computer Vision*. Springer-Verlag, 1995. 2.3, 5.1

[38] LIANG, L., LIU, C., XU, Y.-Q., GUO, B., and SHUM, H.-Y., "Real-time texture synthesis by patch-based sampling," *ACM Transactions on Graphics*, vol. Vol. 20, No. 3, pp. 127–150, July 2001. 2.2

[39] MALIK, J., BELONGIE, S., SHI, J., and LEUNG, T., "Textons, contours and regions: Cue integration in image segmentation," in *International Conference on Computer Vision*, p. II: 918, IEEE Computer Society, 1999. 2.3

[40] MCLACHLAN, G. and KRISHNAN, T., *The EM algorithm and extensions*. Wiley series in probability and statistics, John Wiley & Sons, 1997. 2.3, 5, 5.1

[41] MCMILLAN, L. and BISHOP, G., "Plenoptic modeling: An image-based rendering system," *Proceedings of SIGGRAPH 95*, pp. 39–46, August 1995. ISBN 0-201-84776-0. Held in Los Angeles, California. 2.1

[42] MORTENSEN, E. N. and BARRETT, W. A., "Intelligent scissors for image composition," *Proceedings of SIGGRAPH 1995*, pp. 191–198, Aug. 1995. 3.4, 3.4

[43] PAGET, R. and LONGSTAFF, I. D., "Texture synthesis via a noncausal nonparametric multiscale markov random field," *IEEE Transactions on Image Processing*, vol. 7, pp. 925–931, June 1998. 2.3

[44] PÉREZ, P., GANGNET, M., and BLAKE, A., "Poisson image editing," *ACM Transactions on Graphics, SIGGRAPH*, vol. 22, no. 3, pp. 313–318, 2003. 3.4

[45] POPAT, K. and PICARD, R. W., "A novel cluster-based probability model for texture synthesis, classification, and compression," in *Proc. SPIE Visual Communications*, pp. 756–768, 1993. 2.3

[46] PORTILLA, J. and SIMONCELLI, E. P., "A parametric texture model based on joint statistics of complex wavelet coefficients," *International Journal of Computer Vision*, vol. 40, pp. 49–70, October 2000. 2.2

[47] SAISAN, P., DORETTO, G., WU, Y., and SOATTO, S., "Dynamic texture recognition," in *Proceeding of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. II:58–63, 2001. 2.2

[48] SCHÖDL, A., "Multi-dimensional exemplar-based texture synthesis," *Ph.D. Thesis, Georgia Institute of Technology*, March 2002. 1

[49] SCHÖDL, A. and ESSA, I., "Machine learning for video-based rendering," in *Advances in Neural Information Processing Systems* (LEEN, T. K., DIETTERICH, T. G., and TRESP, V., eds.), vol. 13 of *Proceedings of NIPS Conference*, pp. 1002–1008, MIT Press, 2001. 2.4

[50] SCHÖDL, A. and ESSA, I. A., "Controlled animation of video sprites," in *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pp. 121–127, ACM Press, 2002. 2.4

[51] SCHÖDL, A., SZELISKI, R., SALESIN, D. H., and ESSA, I., "Video textures," *Proceedings of SIGGRAPH 2000*, pp. 489–498, July 2000. ISBN 1-58113-208-5. 2.2, 2.4, 3.5, 3.5

[52] SEDGEWICK, R., *Algorithms in C, Part 5: Graph Algorithms*. Reading, Massachusetts: Addison-Wesley, 2001. 3.1, 3.1.2

[53] SEITZ, S. M. and DYER, C. R., "View morphing: Synthesizing 3d metamorphoses using image transforms," *Proceedings of SIGGRAPH 96*, pp. 21–30, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana. 2.1

[54] SHEWCHUK, J. R., "An introduction to the conjugate gradient method without the agonizing pain," August 1994. 5.3

[55] SHUM, H.-Y. and HE, L.-W., "Rendering with concentric mosaics," in *Proceedings of 26th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 299–306, 1999. 2.1

[56] SHUM, H. and SZELISKI, R., "Construction and refinement of panoramic mosaics with global and local alignment," in *ICCV98*, pp. 953–958, 1998. 2.1

[57] SOATTO, S., DORETTO, G., and WU, Y., "Dynamic textures," in *Proceeding of IEEE International Conference on Computer Vision 2001*, pp. II: 439–446, 2001. 2.2, 3.5, 3.5

[58] SOLER, C., CANI, M.-P., and ANGELIDIS, A., "Hierarchical pattern mapping," *ACM Transactions on Graphics*, vol. 21, pp. 673–680, July 2002. 3.3

[59] SUN, M., JEPSON, A., and FIUME, E., "Video input driven animation (vida)," in *International Conference on Computer Vision*, pp. 96–103, 2003. 2.4

[60] SZELISKI, R. and SHUM, H.-Y., "Creating full view panoramic mosaics and environment maps," *Proceedings of SIGGRAPH 97*, pp. 251–258, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California. 2.1

[61] SZUMMER, M. and PICARD, R., "Temporal texture modeling," in *Proceeding of IEEE International Conference on Image Processing 1996*, vol. 3, pp. 823–826, 1996. 2.2

[62] TONIETTO, L. and WALTER, M., "Towards local control for image-based texture synthesis," *XV Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI)*, pp. 252–260, October 2002. 2.4

[63] WANG, Y. and ZHU, S., "A generative method for textured motion: Analysis and synthesis," in *European Conference on Computer Vision*, June 2002. 2.2

[64] WEI, L.-Y. and LEVOY, M., "Fast texture synthesis using tree-structured vector quantization," *Proceedings of SIGGRAPH 2000*, pp. 479–488, July 2000. ISBN 1-58113-208-5. 2.2, 2.3, 1, 3.5, 3.5

[65] WEI, L.-Y. and LEVOY, M., "Order-independent texture synthesis," Tech. Rep. TR-2002-01, Stanford University CS Department, 2002. 2.3

[66] WEXLER, Y., SHECHTMAN, E., and IRANI, M., "Space-time video completion," in *CVPR 2004*, pp. 120–127, 2004. 2.3

[67] WU, Q. and YU, Y., "Feature matching and deformation for texture synthesis," *ACM Transactions on Graphics (SIGGRAPH 2004)*, August 2004. 2.2

[68] YUAN, L., WEN, F., LIU, C., and SHUM, H.-Y., "Synthesizing dynamic texture with closed-loop linear dynamic system," *European Conference on Computer Vision*, May 2004. 2.4

[69] ZHANG, E., MISCHAIKOW, K., and TURK, G., "Vector field design on surfaces," Tech. Rep. 04-16, Georgia Institute of Technology, 2004. 6.3

[70] ZHANG, J., ZHOU, K., VELHO, L., GUO, B., and SHUM, H.-Y., "Synthesis of progressively-variant textures on arbitrary surfaces," *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 295–302, 2003. 2.4

[71] ZHU, S.-C., EN GUO, C., WU, Y., and WANG, Y., "What are textons," in *European Conference on Computer Vision*, 2002. 2.3